Model-Based Unit Testing using Decision Tables

Peter Verkade

July 2017, 61 pages

UvA supervisor: Ana Oprescu
Host organisation: Axini
Host supervisor: Machiel van der Bijl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Model-based testing is a formal approach to software testing. Model-based testing tools automatically generate high quality test suites based on formal specifications of the tested system. The specification formalisms used by existing tools are based on transition systems, designed for system testing of reactive systems. On the other hand, describing functional specifications used for unit testing with these formalisms is difficult. This thesis presents a new modeling formalism for model-based unit testing, which combines the traditional concepts of decision tables and Hoare logic. A proof-of-concept testing tool is implemented using this modeling formalism. This tool is shown to generate higher quality test suites for source code of the host company compared to the existing, manually designed unit tests, while requiring similar testing effort.

Chapter 1

Introduction

Software testing is a key process in software development to ensure a developed system conforms to its specification. The most common approach to software testing in the software industry is developers and testers manually designing test cases for the developed system. However, building a comprehensive set of test cases covering all aspects of the system is a complex and labor intensive task.

Model-based testing is a formal approach to software testing which solves these problems. Model-based testing revolves around a formal specification of the tested system's behavior called the model. Model-based testing tools use this model to automatically generate a comprehensive set of test cases for the modeled system. These tools deploy testing strategies designed to maximize the test coverage of the described system behavior, resulting in a high quality test suite.

Existing model-based testing tools, such as fMBT [1], TestOptimal [2], and Axini's TestManager [3], are designed for system testing of large software systems. This focus is reflected in the modeling formalisms used by these tools. Their modeling formalisms are based on transition systems, which describe the system's behaviors as a set of system states and transitions between them. The transitions usually represent observable input and output actions of the tested system, which makes these formalisms well suited for modeling reactive systems, systems that interact with the environment.

Unit testing on the other hand tests the behavior of the system on a lower level. The objective of unit testing is to find whether each component implements its specified functionality correctly whereas integration and system testing test whether the components interact with each other correctly. During unit testing, functions and methods are tested against their functional specification, which specify how input of these units relates to their output behavior.

When a modeling formalism based on transition systems is used to describe the behavior of a function or method, the resulting model reflects the structure of the desired computation. However, the intended behavior of the unit, the relation between input and output values, is not modeled explicitly. The input-output behavior described by the model can only be derived by analyzing the possible execution paths, which can be very difficult for more complex specifications. This makes it difficult to detect errors in the model. This is similar to the problems with white-box testing, which are elaborated in section 2.1.2.

[1] 01.org/fmbt
[2] mbt.testoptimal.com
[3] www.axini.com

1.1 Motivating example

This problem is illustrated in the following example. Figure 1.1 shows an implementation of a simple function in the programming language Ruby and a symbolic transition system modeling its behavior. The intention of the function is to calculate exponentiations (limited to positive integer exponents).

```ruby
def exponentiation(base,
    exponent)
  result = base
  while exponent > 0
    result   = result * base
    exponent = exponent - 1
  end
  return result
end
```

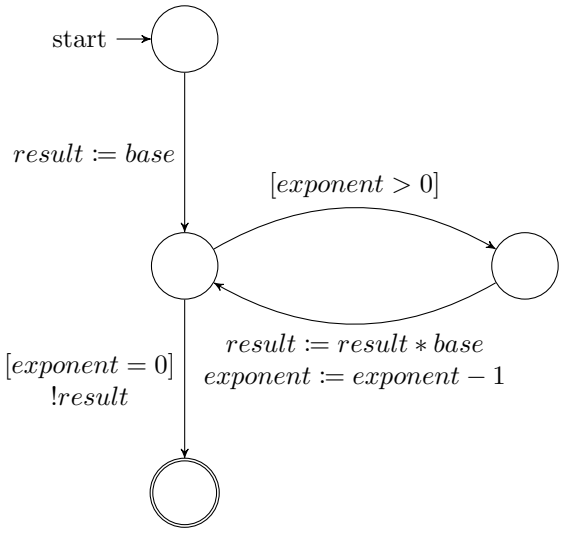

Figure 1.1: Example of a Ruby function and a symbolic transition system modeling its behavior.

If the model on the right were used to unit test the function with a model-based testing tool, the test verdict would be that the implementation conforms to the model. This gives the impression that the implementation is correct. However, after execution, the implementation is found to be incorrect, as it does not compute the correct values. A correct implementation should start with the multiplicative identity `result = 1`. This error was not found as the model does not relate closely to the functional specification of the function, which explicitly states what values it must calculate.

1.2 Research Questions

To perform model-based unit testing effectively, a different modeling formalism is required. Such a formalism should explicitly describe the intended behavior of the modeled functions and methods. To select a good formalism, the behavior of functions and methods which must be modeled must first be identified. From this follows research question 1.

Research question 1: *Which behavior of functions or methods needs to be modeled to allow for model-based unit testing?*

This thesis presents a new modeling formalism for model-based unit testing. This formalism combines the traditional concepts of decision tables and Hoare logic and applies them to model-based unit testing. Decision tables are a traditional format to describe conditional behavior in a compact form. This description format must be related to the identified behavior of functions and methods to be modeled. From this follows research question 2.

Research question 2: *How can the behavior of functions and methods be modeled using decision tables?*

To show the feasibility of this new modeling formalism, a proof-of-concept testing tool is implemented which uses this modeling formalism for model-based unit testing. From this follows research question 3.

Research question 3: *How can decision tables be used in model-based testing?*

Since model-based testing involves a number of steps, this research question is expanded into three subquestions.

Research question 3a: *How can decision table theory be used to automatically check decision table specifications?*

Research question 3b: *How can test cases be generated automatically from decision table specifications?*

Research question 3c: *How can optimal test suites be selected from decision table specifications?*

Furthermore, a commonly used feature of decision tables is the ability to automatically optimize their size. From this follows research question 4.

Research question 4: *how can decision table theory be used to automatically minimize the size of decision table specifications?*

Finally, this model-based unit testing approach is compared to the traditional practice of manual unit testing. This is done by testing existing source code of the host company Axini. This experiment shows that the testing tool generates higher quality test suites compared to the existing, manually designed unit tests. It also shows that this unit testing approach provides this benefit without increasing testing effort. The research question corresponding to this experiment is research question 5, which is expanded into two subquestions.

Research question 5: *How does model-based unit testing improve software testing compared to manual unit testing?*

Research question 5a: *How does model-based unit testing affect the quality of software testing?*

Research question 5b: *How does model-based unit testing affect the effort required for software testing?*

1.3 Contributions

This thesis presents the following contributions.

1. A new modeling formalism for model-based unit testing is presented. This modeling formalism is motivated and supported by theory.
2. The implementation of a testing tool based on this modeling formalism is discussed, which shows the feasibility and required technology of such a tool.
3. A new testing strategy is presented, which allows optimal test suites to be generated from decision table specifications. It is shown how this strategy can be implemented efficiently.
4. The model-based unit testing approach is compared to the traditional practice of manual unit testing. This comparison shows that this new approach is indeed an improvement over the traditional practice.

1.4 Project context

This graduation project is performed in cooperation with Axini, a software company specialized in model-based testing. Their main product is TestManager, a model-based testing tool which uses symbolic transitions systems to model systems under test.

1.5 Thesis outline

This thesis is structured as follows. First, chapter 2 presents the required background knowledge on software testing and traditional model-based testing. Chapter 3 presents the theory behind the presented modeling formalism and motivates why the formalism works for model-based unit testing. This chapter answers research questions 1, 2 and 4. Chapter 4 shows how all key steps of the traditional model-based testing process can be implemented for this formalism, including model-checking, test case generation, test case selection and test case execution. This answers research question 3. Chapter 5 describes the experiment in which the model-based unit testing approach is compared with manual unit testing. Chapter 6 contains the discussion of the experiment results, which answers research question 5. Related work on decision table modeling and automated unit testing is discussed in chapter 7. Finally, chapter 8 presents the conclusions of this thesis.

Chapter 2

Background

This chapter describes required background knowledge on sofware testing and model-based testing. It presents the transition system based modeling formalisms used by traditional model-based testing tools. Finally, related work on decision table testing and automated unit testing is discussed.

2.1 Software testing

The objective of software testing is twofold: to find errors in the tested system, or to gain confidence that the system is correct. Software testing is an important activity during software development required to verify that a developed system conforms to its specification. The software system subjected to testing activities is usually referred to as the *system under test* (SUT).

Software testing is performed by executing *test cases* against the SUT. A test case consists of input for the SUT and the expected output. The output from the SUT can have many forms, such as an observable action, an internal state change, or the result of a computation. When a test case is executed against the SUT, the output of the SUT is compared to the expected output described in the test case. A *test suite* is a collection of test cases which together are deemed sufficient to gain confidence in the correctness of the SUT. A test suite aims to cover as many aspects or behaviors of the SUT as possible.

To gain complete confidence in the SUT, a test suite should contain a test case for every possible input value to show that the system always operates correctly. However, even for simple systems, the amount of possible test cases is almost always very large, and in some cases even infinite. To test the SUT in an acceptable amount of time, an optimal test suite consists of only a minimum amount of test cases required to cover each aspect or behavior of the SUT. This selection of test cases is one of the main challenges of software testing.

2.1.1 Types of software testing

Software testing is performed in different ways at various stages of software development. *Unit testing* tests a software component in isolation against a functional specification of its interface. *Integration testing* tests multiple software components and their interactions. *System testing* tests the functionality of an entire software system including all its components. Finally, *acceptance testing* validates whether the developed system meets fulfills the customers functional and non-functional requirements.

The V-model shown in figure 2.1 is one way to model the software development lifecycle. The model clearly shows how the different types of testing are used to verify the results of different development phases. Since the testing types have different purposes, they use different specifications of the desired behavior as starting point.

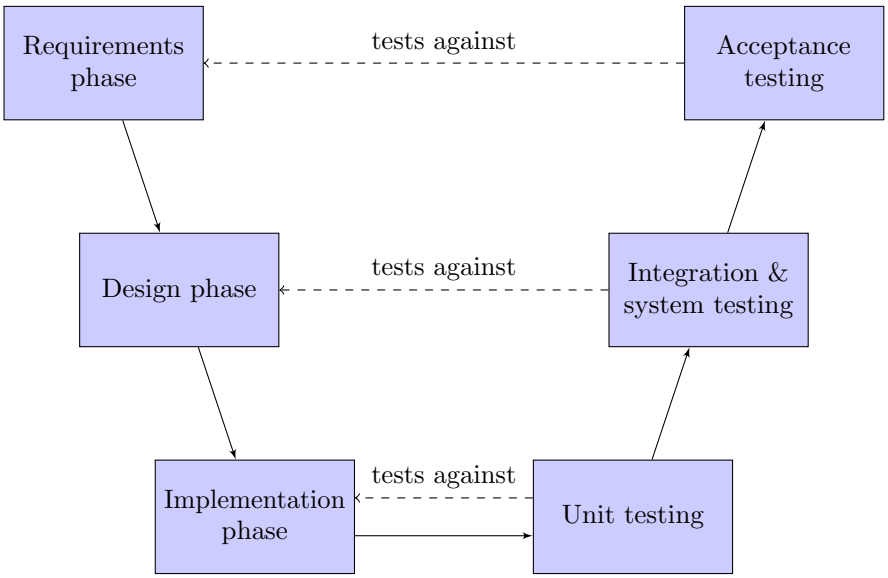

Figure 2.1: The V-model of the software lifecycle describes how the different types of software testing are used to verify the results of different development phases.

2.1.2 Approaches to software testing

Software testing makes a distinction between two major approaches, *black-box* and *white-box testing*. These two approaches differ in how they create and select test cases for the SUT.

Black-box testing

Black-box testing, uses only a specification of the SUT's input-output behavior to create test cases. The SUT is regarded as a black box, and its internal structure is not observed during test case creation and selection. To use black-box testing, the specification of the system's behavior must be very clear. Missing or ambiguous use cases result in an incorrect or incomplete test suite. In software development, the intended behavior of a system or component is often documented using natural language, which can make black-box testing difficult to apply in practice.

White-box testing

White-box testing does observe the internal structure of the SUT, usually in the form of source code, to find and test the distinct behaviors of the system. Test cases are created and selected to test whether each observed behavior of the SUT is implemented correctly. Furthermore, white-box testing allows the tester to identify and test error sensitive execution paths, which is very difficult to do in black-box testing.

When the intended behavior of the SUT is documented well, white-box testing can be combined with black-box testing. In that case, relevant input-output test cases can be extended with test cases that explicitly cover error sensitive execution paths identified in the implementation. However, when the intended behavior of the SUT is insufficiently specified, white-box testing has inherent risks [MSB04, p. 13]. While test cases can be created to test the behaviors observed in the SUT, no test cases can be designed to cover desired behavior that is neither documented nor implemented. This can result in missing test cases and undetected errors. Furthermore, when the intention of the system is not documented clearly, the tester will have to make assumptions about the desired behavior. This can cause the tester to interpret incorrectly implemented behavior as intended.

2.1.3 Testing strategies

Since testing all possible test cases is unrealistic for most systems, selecting a minimum amount of test cases required to gain confidence in the correctness of the SUT is an important part of the testing procedure. Various testing strategies exist to reduce the number of test cases to a minimum. This section describes two testing strategies relevant to the modeling formalism presented in this thesis: *equivalence partitioning* and *boundary value analysis*. Both of these strategies are forms of black-box testing, using only the specification of the system to select test cases.

Equivalence partitioning

The equivalence partitioning strategy reduces the number of test cases by grouping test cases which elicit the same behavior of the SUT. The goal of this testing strategy is to test whether each behavior of the SUT is implemented correctly. Thus, only a single test case is required from each partition [MSB04].

For example, if a system has three distinct behaviors for positive, negative and zero value inputs, the equivalence partitioning strategy will group all test cases with positive input values into a partition, and the same for the cases with negative and zero values. Then, only a single test case is selected to form each partition. Other test cases are discarded. The remaining three test cases are sufficient to test whether or not the SUT implements all three distinct behaviors.

Boundary value analysis

The equivalence partitioning strategy test whether each behavior of the SUT is implemented. However, it does not test whether all input values elicit the correct behavior of the SUT. The boundary value analysis (BVA) strategy expands on the equivalence partitioning strategy by including this aspect in test case selection [MSB04].

As discussed before, testing all input values of the SUT is in most cases unrealistic. Therefore, BVA limits itself to the cases which are most likely to be implemented incorrectly: the edges of the input domains for each behavior. Input values at the edge of an equivalence partition are more likely to be implemented incorrectly, due to so-called off-by-one errors [Fos80]. The boundary value analysis (BVA) strategy takes this into account by extending the equivalence partitioning test cases with additional test cases for these boundary values.

In its most basic form, BVA adds additional test cases for the minimum and maximum possible value of each input variable. BVA can be extended by also testing values near the boundaries of the domains, such as one above the minimum value and one below the maximum value. However, these cases are less likely to fail then the boundaries themselves, as they no longer represent off-by-one errors. BVA can also be extended by testing values just outside the domain, such as one below the minimum value and one above the maximum value. However, when the input domain is fully partitioned, these values are boundary values of other partitions as well. As a result, they are already included as test cases for those partitions.

When the input to the SUT consists of multiple values, BVA can be applied to each of them. The resulting boundary values can be combined into test cases in multiple ways. The simple strategy is to select one test case for normals values plus one for each boundary value, which results in $2n + 1$ test cases for n input values. It is also possible to select all combinations of the minimum, maximum, and a normal value of each input, which results in 3^n test cases for n input values. The first option is often preferred for scalability whereas the second provides more thorough testing.

2.2 Model-based testing

Model-based testing is a formal approach to software testing. In model-based testing, the specification of the SUT is described in a formal model. The testing tool uses this model to derive relevant test cases, and select test cases in such way that they optimally cover the different aspects or behaviors of the SUT described in the model. The model can also be used to check certain properties of the system. For example, the transition system models presented in section 2.2.2 allow the testing tool to check the reachability of certain system states.

Model-based testing forms a contrast to the common industry testing practice of developers or testers manually designing and selecting test cases. Model-based testing has numerous benefits over this manual approach to testing.

Automation: Model-based testing allows for a greater degree of automation by automating test case generation and selection in addition to test execution and test result evaluation.

Test quality: Model-based testing tools can generate test suites consisting of a minimal amount of test cases to optimally cover the behavior of the SUT described in the model.

Model-checking: The model of the SUT allows for formal reasoning about its properties. For some types of models, it is possible to check whether the SUT can reach an invalid system state.

Maintainability: When the specification of the SUT changes, model-based testing requires only the model to be updated, whereas manually written tests all have to be checked and updated one by one.

Documentation: The model of the SUT also serves as a form of documentation describing the system's behavior.

2.2.1 The model-based testing process

Figure 2.2 shows an overview of the model-based testing process. The model is a formal description of the system's specification. The model is checked and used to generate an optimal test suite for the SUT. This test suite is then used by the test execution component, which checks whether the SUT performs the expected behavior described by the test cases. To do so, the execution components communicates the input described in the test cases to the SUT and verifies whether the SUT gives the expected output.

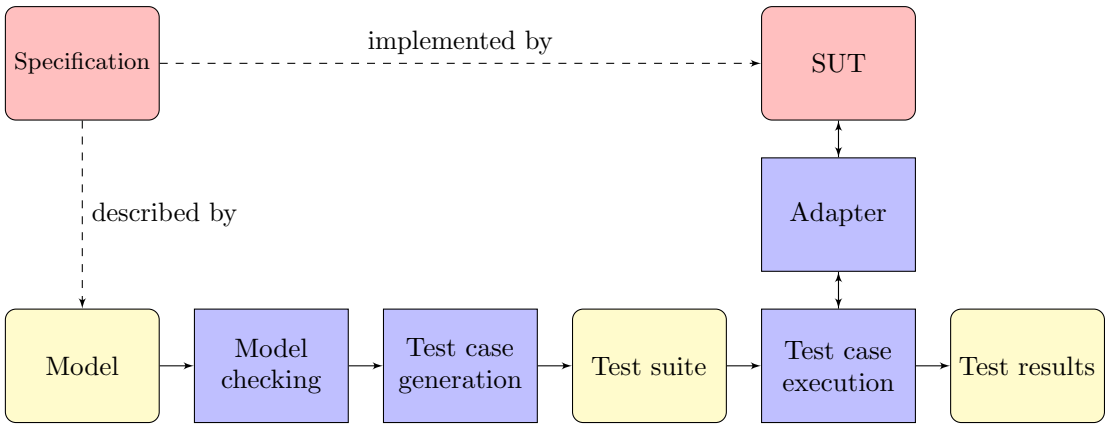

Figure 2.2: Overview of the model-based testing process. Development artifacts are shown in red. Testing artifacts are shown in yellow. The software components in blue are implemented by a model-based testing tool.

Communication with the SUT is not generic, as different tested systems can use different communication channels or formats. To make the communication with the SUT generic, communication to and from the SUT goes through an adapter component. The adapter is usually custom made for each SUT, and hides its communication details behind a generic interface.

The standard process shown in figure 2.2 separates test case generation and execution. This approach is known as offline testing. Some approaches to MBT combine test case generation and execution by dynamically extending generated test cases based on the responses of the SUT. This is known online testing. This approach could be used to test non-deterministic systems, for which no full input-output sequences can be generated upfront.

2.2.2 Transition system based modeling formalism

Most existing model-based testing tools, such as fMBT, TestOptimal, and Axini's TestManager, use a modeling formalism based on transition systems. This section gives a concise overview of such modeling formalisms and how they are used to model system behaviors.

Labeled transition systems

A *labeled transition system* (LTS) is a structure consisting of states and labeled transitions [Tre08]. A labeled transition system is defined as a four-tuple $< Q, L, T, q_0 >$ where Q is a non-empty set of states, L is a set of labels, T is a set of transitions and q_0 is an element in Q which acts as the starting state of the system. A transition is defined as a triple $< q, \mu, q' >$ where q is the origin state , q' is the target state and μ is a label from L associated with the transition. The label μ may also be the unobservable τ label, which is similar to an ϵ transition in automata theory.

Transitions represent actions which can be taken to reach different system states. A transition $< q, \mu, q' >$ can be read as "in state q, the system can take action μ to reach state q'". The τ label is reserved for internal actions, which are unobservable to the outside world.

Two special kind of action labels can be used in a labeled transition systems: input and output labels. These donate input and output actions between the system and its environment. Input actions are denoted by a ?-symbol and output actions are denoted by a !-symbol. These actions make transitions systems a suitable formalism to model the behavior of *reactive systems*, systems which react to the environment. Labeled transition systems naturally describe which input and output actions are available in each system state and how the system state is changed when they are performed.

Figure 2.3 shows an example of a labeled transition system that models the functionality of a simple coffee machine with buttons to select milk and sugar and to receive coffee. This example is adapted from Lamberink [Lam14].

Symbolic transition systems

Realistic systems usually have a massive amount of valid states they can be in. As a consequence, modeling these states and their relations in a labeled transition system can be a daunting task. The example in figure 2.3 shows that two boolean variables require at least four dedicated states with corresponding transitions to represent all possible combinations of values. Moreover, unbounded variables such as natural numbers could theoretically require an infinite amount of states. Even when they are limited to a bounded domain, the amount of states required to model all combinations of values of variables becomes immense.

This problem is a result of the fact that states in labeled transition systems have a double purpose. They both represent control flow states of the system as well as values of internal variables. To solve this, internal variables can be separated from the control flow states. This results in *symbolic transition systems* (STS). Symbolic transition systems are the modeling formalism used by Axini to perform model-based testing.

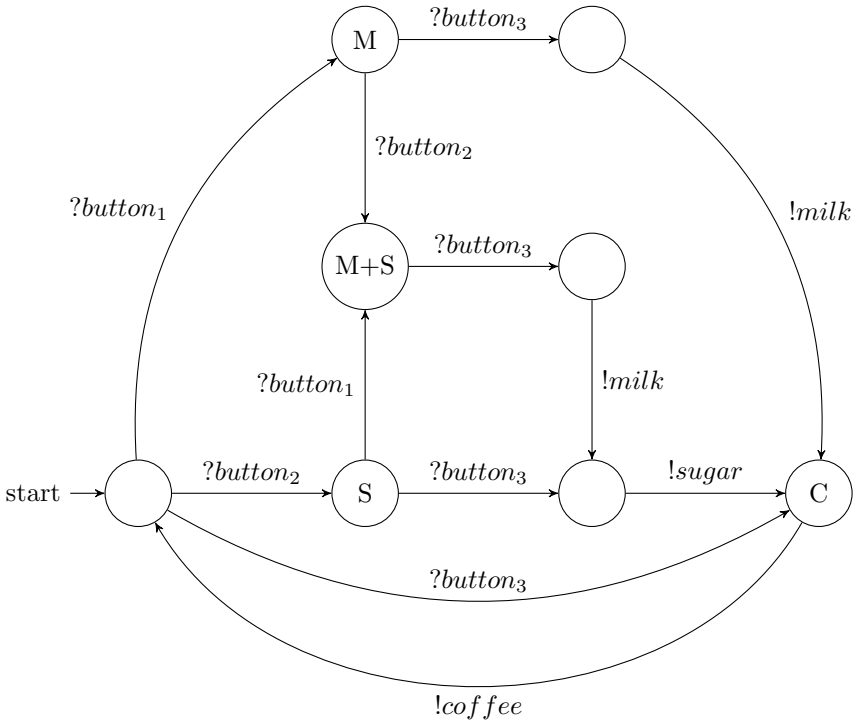

Figure 2.3: Example of a labeled transition system used to model a simple coffee machine. The input actions ?$button_1$ and ?$button_2$ are used to select milk (M) and sugar (S) respectively. The input action ?$button_3$ triggers the dispense of coffee (C) with the selected flavorings.

A symbolic transition system is defined as the five-tuple $< Q, L, T, V, q_0 >$. Q, L, T and q_0 represent the states, labels, transitions and starting state similarly to an LTS. The first addition is the set V, which contains the internal variables used in the STS. These variables are used in actions from the labels L.

Furthermore, symbolic transition systems allow variables to be updated during transitions. This is described by the labels on transition. Updating variables is usually notated as $v := e$, where v is a variable from V and e is a value or expression assigned to the variable. Linear sequences of variable assignments are sometimes notated as a single transition for brevity. Furthermore, labels can have guard expressions notated as $[e]$, which means that these action can only be performed when the expression holds for the current variable assignment.

Figure 2.4 shows a symbolic transition system modeling the behavior of the same coffee machine modeled in figure 2.3. The variables *milk* and *sugar* no longer require distinct states to represent their values. Instead, they are now internal variables updated on transitions. Guard expressions are used to determine whether or not to perform the output actions !*milk* and !*sugar*.

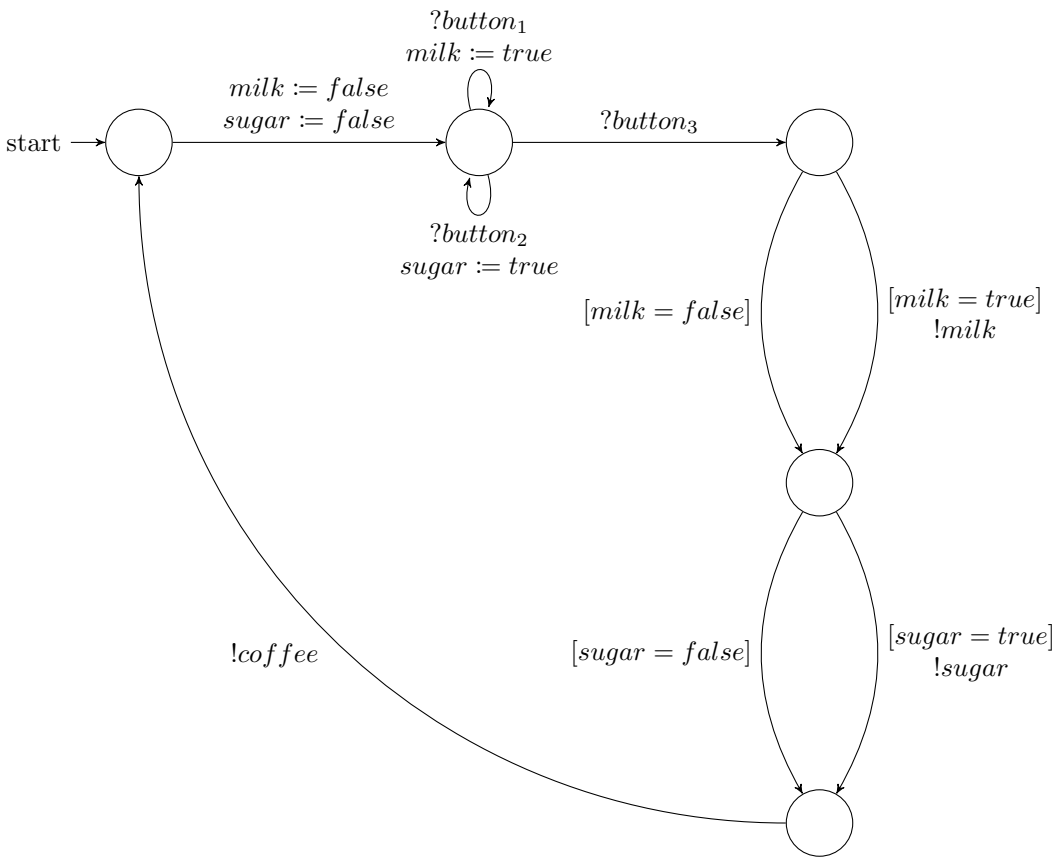

Figure 2.4: Example of a symbolic transition system used to model the same simple coffee machine modeled in the labeled transition system in figure 2.3.

Chapter 3

Description of the modeling formalism

This chapter describes the theory behind the presented modeling formalism. First, the behavior to be described by a modeling formalism for model-based unit testing is discussed. Then, decision tables are introduced and decision table theory is presented. The behavior of mathematical functions and programming functions is discussed, and it is shown how their input-output behavior can be modeled using Hoare logic. Finally, the relation between Hoare logic and decision tables is shown, and the advantages of encoding Hoare logic in decision tables are discussed.

3.1 Function and method behavior to be modeled

In traditional model-based testing, the testing tool interacts with the SUT by communicating input and output. This communication interface of the SUT forms the behavior that can be tested. The conformance of the SUT to the model is determined solely by testing whether the actual input and output of the SUT are described in the model. Model-based testing can only be applied to unit testing when a similar input-output interface can be identified for functions and methods.

In unit testing, methods and functions are tested in isolation. To interact with these systems, they are executed directly by the testing tool. Their input arguments and returned output values are the obvious ways to communicate input and output. However, the testable interface of functions and methods is much broader. Output behavior of functions and methods can take many forms, such as assigning new values to global variables, reading and writing on `stdin` and `stdout`, creating or altering files and raising errors and exceptions. All such behaviors are part of the unit's response to the input given by the testing tool.

Furthermore, when a (non-static) method of a class is tested, an instance of that class must be created as test subject. The values of the instance variables of that object can be used inside the method as well, effectively acting as an additional input channel. Moreover, the values of these variables can be updated during execution of the method, which makes them an output channel as well.

All these possible input and output channels of a function or method allow a model-based testing tool to interact with them. A modeling formalism for model-based unit testing must therefore be able to describe (large parts of) this interface.
Additionally, transition systems also model the relation between input and output behavior of the SUT through changes in system state. This allows the model to describe which input and output can be expected after a previous sequence of input and output. For almost all functions and methods, the executed behavior is determined by the given input. Thus, to model the behavior of functions and methods, the modeling formalism should not only list the possible input and output, but also describe how they relate to each other.

3.2 Decision tables

This section describes decision tables and decision table theory. It will present the concept of decision tables and two important decision table notations. This section also includes algorithms from decision table theory to check two important properties of decision tables and to reduce their size when possible.

Decision tables describe decision situations, in which different actions must be taken based on a number of conditions. The different cases are described in *decision rules*, which form the basis of decision tables. Decision rules have a *if-then* structure, and describe the actions which must be taken when the rule's conditions are met. Decision rules are used in various scientific fields, such as game theory, for example to describe an optimal strategy to play a game. They can also be used in many other settings, such as describing business rules within a corporate setting.

Decision tables are a compact tabular notation for describing a sets of decision rules. Decision tables consist of four quadrants, which respectively describe the relevant conditions, the condition entries, the relevant actions and the action entries. The decision rules are encoded in the columns of the decision table covering both condition and action entries. Table 3.1 shows an overview of this general decision table structure.

conditions	condition entries
actions	action entries

Table 3.1: Overview of the general decision table structure.

Table 3.2 shows a simple example of a decision table describing which apparels to bring given the weather conditions outside. Each column encodes a decision rule describing a distinct weather condition and the relevant actions to be taken. Together, these decision rules fully describe the decision situation.

Rain	no	no	yes	yes
Strong wind	no	yes	no	yes
Wear a jacket	-	X	X	X
Bring an umbrella	-	-	X	-

Table 3.2: Example of a simple decision table.

Conditions in the decision table can be expressed as relations or predicates on variables. An input to a decision table is an assignment of variables to values which allows the conditions to be evaluated. Such an assignment of variables to values is called a *transaction*. In the example above, the conditions *rain* and *strong wind* can be seen as boolean variables. To make a decision based on the decision table, these variables must be given a value.

Decision tables can be written using various different forms and notations. One important distinction in decision table notations is how the condition and action entries are notated. The two major variants, *limited entry* and *extended entry* decision tables will be shown in the following subsections. *Mixed entry* decision tables allow rows with either limited or extended entries, but do not allow the different entries to be mixed in the same row.

3.2.1 Limited entry decision tables

Limited entry decision tables are table which use a small set of symbols as condition and action entries to represent the relation between conditions and actions in each decision rule. These symbols tell whether conditions must evaluate to true or false, and whether actions must be executed or not.

The allowed conditions entries are true (`T`) or yes (`Y`) for when the condition must hold and false (`F`) or no (`N`) for when the condition must fail. Limited entry decision tables can also use the immaterial entry (`-`). This entry has two meanings. It can either mean that the condition may evaluate to both true and false or that the condition is irrelevant or impossible within that decision rule. Action entries are limited to execute (`X`) and don't execute (`-`). The decision table in table 3.2 describing the weather conditions is an example of a limited entry decision table.

Each condition can only evaluate to true or false for a given transaction. As a result, a decision table with n condition can have a maximum of 2^n decision rules. The actual number of decision rules is often lower as combinations of evaluations may not be possible and decision rules with the `-` entry replace both the rules with true and false for the corresponding condition.

3.2.2 Extended entry decision tables

Extended entry decision tables use a more compact notation by using binary relations on transaction variables as entries. This allows multiple conditions to be expressed in a single extended entry row. The following relations are commonly used: $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$. The $=$ relation is often simplified further to the value the variable must be equal to.

An example of an extended entry decision table is shown in on the left in table 3.3. The table contains the same decision logic as the limited entry table shown on the right.

$input_1$	> 0	> 0	$= 0$	< 0
$input_2$	≥ 0	< 0	-	-
$action_1$	X	-	X	X
$action_2$	-	X	-	X

$input_1 > 0$	T	T	-	-
$input_1 = 0$	-	-	T	-
$input_1 < 0$	-	-	-	T
$input_2 \geq 0$	T	F	-	-
$action_1$	X	-	X	X
$action_2$	-	X	-	X

Table 3.3: Example of an extended entry decision table on the left. The decision rules in this table are identical to the ones in the limited entry table on the right.

The extended entry decision table notation is often used when the table contains many similar conditions on a small set of variables. As seen in the example above, the amount of condition rows is reduced significantly in the extended entry variant.

Extended entry decision table can always be transformed into limited entry decision tables. To do so, each row is expanded into a series of limited entry rows which each condition the relation as the condition. The condition entries for the new row become `-`, except for the column from which the relation originated, which becomes `T`. When a row with an equivalent condition was already present, the `T` entry can also be inserted into that row instead. When a row with an opposite condition was already present, a `F` entry can be inserted into that row instead.

3.2.3 Decision rules as propositional logic

Decision rules can be expressed as formulas in propositional logic. Each decision rule has the form of an implication in which the right evaluations of the conditions together imply the execution of the actions.

The conditions of the decision rule are combined into a conjunction, as all must hold for that decision rule to be used. When the decision rule requires a condition to evaluate to false (`F`), the negation of that condition is used in the conjunction instead. Conditions with a `-` entry are not included into the formula, as both evaluations are allowed and the disjunction ($condition \vee \neg condition$) forms a tautology.

The actions of the decision rule are represented by propositional atoms stating the execution of the actions. A negation of such atom represents that the action is not executed. These atoms are combined into a conjunction as well, as they are all implied by the decision rule.

As an example, the four decision rules from the decision tables in table 3.3 can be mapped to the following four formulas.

$$
\begin{array}{lcllcl}
(input_1 > 0) & \wedge & (input_2 \geq 0) & \Longrightarrow & action_1 \wedge \neg action_2 \\
(input_1 > 0) & \wedge & \neg(input_2 \geq 0) & \Longrightarrow & \neg action_1 \wedge action_2 \\
(input_1 = 0) & & & \Longrightarrow & action_1 \wedge \neg action_2 \\
(input_1 < 0) & & & \Longrightarrow & action_1 \wedge action_2
\end{array}
$$

3.2.4 Decision table checking algorithms

Decision tables can be checked systematically for two important properties: *completeness* and *unambiguity*. A decision table is complete when it contains a decision rule for every possible transaction. A decision table is unambiguous when every possible transaction only matches with a single decision rule. When a decision table has both these properties, it forms a valid description of the decision situation.

Checking these two properties requires domain knowledge of the context of the decision table. This will be illustrated in the following example. Observe the decision table in table 3.4. It describes business logic to determine the discount given to a certain customer based on their customer type and age. The table appears to be incomplete, as the decision rules do not describe gold customers younger than 65 and silver customers older than 65. However, in this business context, it is possible that silver customers are upgraded to gold customers when they reach the age of 65. In that case, the decision table describes all possible customers. On the other hand, if a customer type exists that is neither gold, silver or basic, for example a platinum customer, the decision table would again be considered incomplete, as such a customer is not described.

Sensible conclusions about the completeness of the decision table cannot be reached without knowledge of the context of the conditions and variables. To check the completeness of this decision table, knowledge about the possible customer types and their relation with customer age (if any) is required.

customer type customer age	gold ≥ 65	silver < 65	basic ≥ 65	basic < 65
discount	30%	20%	10%	0%

Table 3.4: Example of a decision table describing business logic regarding discount received by certain customers.

Such domain knowledge can be encoded and automated using modern tools. The testing tool presented in this thesis uses a finite domain constraint solver to reason about conditions and variable values. This is described in chapter 4. Implementing the required domain knowledge results in an oracle which tells whether a given set of conditions allows for valid transactions or not. The algorithms for checking the completeness and unambiguity properties presented in this section make use of such an oracle.

The presented algorithms are only applicable on limited entry decision tables. As discussed before, extended entry and mixed entry decision tables can always be transformed into limited entry decision tables to enable the use of these algorithms.

Checking completeness

As discussed before, the maximum amount of decision rules of a table with n conditions is 2^n. To check whether a decision table is complete, all 2^n possible decision rules are generated. Each of these rules is checked whether the rule is possible and contained in the decision table. The action entries are not relevant for the checking algorithm.

As an example, observer the decision table in table 3.5. To check whether this table is complete all $2^3 = 8$ possible decision rules are generated. An oracle is provided which knows that $input_1$ and $input_2$ can both be chosen from $\mathbb{N}$.

$input_1 > 0$	T	F
$input_1 = 0$	F	T
$input_2 > 0$	-	T
$action_1$	X	-
$action_2$	-	X

Table 3.5: Example of an incomplete decision table.

The decision rule `F, T, T` is directly included in the decision table.

The decision rules `T, F, T` and `T, F, F` are indirectly included, as they are both contained in the first decision rule `T, F, -` since the `-` entry represents both `T` and `F`.

The decision rules `T, T, T` and `T, T, F` are not included. However, no value of $input_1$ can satisfy both the conditions $input_1 > 0$ and $input_1 = 0$ at the same time.

The decision rules `F, F, T` and `F, F, F` might be missing from the decision table. The oracle is used asses whether a these decision rules are possible. In this case, the oracle knows that $input_1$ cannot take a negative value, so these decision rules cannot be possible.

The final decision rule `F, T, F` is not contained in the decision table. However, a transaction with $input_1 = 0$ and $input_2 = 0$ would match on this decision rule. Since $0 \in \mathbb{N}$, the oracle knows that such a transaction is possible. Thus, the decision table is found to be incomplete. Without the oracle, this could not have been decided.

Checking unambiguity

Checking whether a decision table is unambiguous follows a similar procedure as checking for completeness. Two decision rules are directly mutually exclusive when one rule requires a condition evaluate to true where the other rule requires it to evaluate to false. When this is not the case, they might match on the same transaction. To test for this, a new decision rule is created which encompasses both rules. This is always possible since none of the condition entries contradict. Then, the oracle is used to determine whether any transaction exists for this new rule. If any exists, that transaction would match both original rules, which is proof of ambiguity. A table is unambiguous if no pair of rules is ambiguous.

As an example, observe the decision table in table 3.6. Decision rules 1 and 3 are mutually exclusive, as they require a different evaluation of the first condition $input > 0$. The same is true for the pair of rules 2 and 3. However, the rules 1 and 2 do not have any contradicting conditions. To test whether these rules are ambiguous, their are combined into a new decision rule that matches both: `T, T`. When the oracle declares a positive value for both $input_1$ and $input_2$ to be a valid transaction, the decision table is found to be ambiguous.

$input_1 > 0$	T	-	F
$input_2 > 0$	-	T	F
$action_1$	X	-	X
$action_2$	-	X	X

Table 3.6: Example of an ambiguous decision table.

3.2.5 Decision table optimization algorithms

Decision tables sometimes contain redundant information, which allows them to be rewritten in a smaller form. This makes the table easier to read and maintain. This simplification of decision tables can also be performed systematically, which allows the process to be automated. The simplification algorithm is based on two transformations, which are performed in relevant situations.

1. Combining two decision rules using an immaterial (`-`) entry
2. Combining two condition rows when they express the same information

Again, the described algorithms are only applicable to limited entry decision tables.

Combining decision rules

Observe the left decision table shown in table 3.7. The table contains all $2^2 = 4$ possible decision rules. However, the first and second decision rules are almost identical. They can be merged into a new decision rule using an immaterial (`-`) entry to represent both the true (`T`) and false (`F`) evaluations of the second condition. The resulting decision table is shown on the right.

Optimizable table

$condition_1$	T	T	F	F
$condition_2$	T	F	T	F
$action_1$	X	X	-	X
$action_2$	-	-	X	X

Optimized table

$condition_1$	T	F	F
$condition_2$	-	T	F
$action_1$	X	-	X
$action_2$	-	X	X

Table 3.7: Example of a decision table which can be optimized by combining decision rules.

A pair of decision rules can be combined when they meet two conditions. First of all, the decision rules must map to the same actions to be executed, since different sets of actions can only be represented by different rules. Furthermore, their condition entries may only contradict on a single condition. The new decision rule will be mostly identical to the original decision rules, but will contain an immaterial entry for the contradicting condition.

If the decision rules were to contradict on two conditions, a combined decision rule with immaterial entries for both these conditions would represent four decision rules instead of the original two. Sometimes four decision rules can indeed be combined into one. This effect can be achieved by repeating this optimization step until a fixed point is reached.

This optimization on decision rules can also be reversed. This allows an optimized decision table to be expanded into a larger decision table containing decision rules using only `T` and `F` condition entries.

Combining conditions

When describing all relevant conditions for a decision situation, it possible to find conditions which are equivalent or each others negation. The resulting decision table then contains redundant information, as these pairs of conditions can be represented by only one of them.

Observe the left decision table in table 3.8. The conditions $input_1 > 0$ and $input_1 \leq 0$ are each others opposite. As the second condition is the negation of the first, we can express a truth value for the second condition as a false value for the first. The same holds for the third and fourth decision rules. The resulting decision table is shown on the right.

Optimizable table			
$input_1 > 0$	T	-	-
$input_1 \leq 0$	-	T	T
$input_2 > 0$	-	T	-
$input_2 \leq 0$	-	-	T
$action_1$	X	-	-
$action_2$	-	X	-
$action_3$	-	-	X

Optimized table			
$input_1 > 0$	T	F	F
$input_2 > 0$	-	T	F
$action_1$	X	-	-
$action_2$	-	X	-
$action_3$	-	-	X

Table 3.8: Example of a decision table which can be optimized by combining conditions.

In this case, the equivalence or negation of conditions can be assessed using the semantics of the relational operators. In the general case, the oracle containing domain knowledge can be used to determine such a relationship between conditions. This allows the optimization step to be performed even when the conditions might not appear to lend them for this purpose.

To do so, the oracle is used to determine whether a valid transaction can be created for the following pairs of conditions: $(condition_1 \wedge condition_2)$, $(\neg condition_1 \wedge condition_2)$, $(condition_1 \wedge \neg condition_2)$ and $(\neg condition_1 \wedge \neg condition_2)$. If valid transactions are only possible for the pairs in which both conditions are true or false, the conditions must be equivalent (within the context of the domain). When only the pairs in which one condition is true and the other is false lead to valid transactions, the conditions must be opposite.

3.3 Functions and program units

This section gives a definition of mathematical functions and shows how their input-output behavior is similar to that of executable program units.

In programming, the word function has been adopted to refer to subroutines which return computed values. To eliminate confusion, for the rest of this chapter, the term *program unit* will be used to refer to procedures, subroutines, methods or other reusable units of computation used in programming, whereas the term function will be used exclusively for the mathematical objects. In subsequent chapters, the context will make clear what is referred to with the term function.

3.3.1 Functions

A function is a mapping from input to output, in which each input is mapped to exactly one output. The input of a function is referred to as the *argument* of the function, The output is referred to as the *value* of the function. The set of all possible arguments is called the *domain* of the function. The set of all possible values is called the *range* of the function. A function is usually expressed as a relation from its domain to its range.

Some functions can be represented with an algebraic expression or algorithm. These function are called *computable*. However, any relation from one set to another with the property that each element is mapped to exactly one other represents a function. Figure 3.1 shows a graphical representation of a simple function which maps the domain consisting of six letters to a number. This function will be used as the leading example in section 3.4.

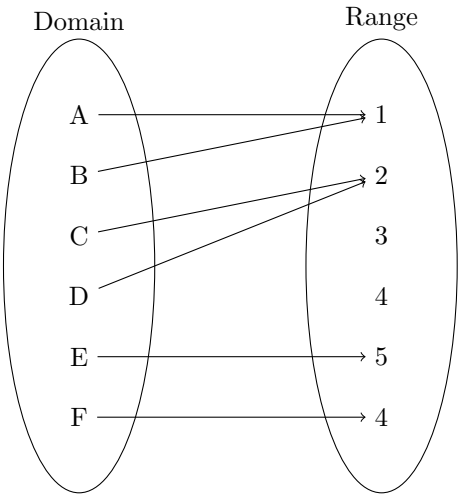

Figure 3.1: Graphical representation of a function. A function maps each element in the domain to exactly one element of the range.

3.3.2 Program units

Program units often perform a similar mapping from input arguments to output values as functions. This mapping is expressed as a computation to be executed by the machine. However, program units are able to do more than just this mapping, as they can perform actions side-effects. Examples of side-effects are changing a global or instance variable, creating a new file, or sending output to `stdout`. Because of this, the behavior of program units can be seen as an extension of the input-output behavior of functions.

This section will motivate how the input-output behavior of functions can be modeled using Hoare logic. This forms the basis for the presented modeling formalism for model-based unit testing. The modeling formalism is also able to describe the additional behavior of program units. This will be discussed in section 3.5. For now, modeling the shared input-output behavior of functions and program units has the focus.

3.3.3 Function specifications

The objective of the presented modeling formalism is to describe the input-output behavior of functions and by extension of program units. For model-driven engineering, in which the model is used as specification for implementing the system, a highly detailed model of the SUT is desirable. On the contrary, for the specific purpose of software testing, the modeling formalism must strike a balance between being too detailed and too abstract.

On one hand, the specification should not be too detailed. When the described function is computed by a complex expression or algorithm, modeling the behavior precisely requires an equally complex model. Such models would have two major practical problems. First of all, modeling a program unit would be equally difficult and labor intensive as implementing it. This would massively increase testing effort. Moreover, when the implementation does not conform to the model, both are equally likely to be incorrect, as they are similarly complicated. These two problems would make such a modeling formalism impractical for model-based testing.

The desired formalism should instead lower the complexity of the model by abstracting the behavior of the modeled function. Such an abstraction would describe parts or properties of the mapping from arguments to output values. This mapping of arguments to values can be abstracted to various different levels of detail. However, when the specification is too abstract, too many functions would conform to the specification. In practice, it would be possible for an incorrect implementation to meet the same specification as the desired implementation. To prevent this, the model should be sufficiently detailed to distinguish different functions.

To illustrate what detail means in the context of specifying function behavior, the *sine* function is used as an example. On the most basic level, the sine function maps real numbers to other real numbers. Such a specification already distinguishes it from other functions such as the one shown in figure 3.1. This property is usually written as follows.

$$\sin : \mathbb{R} \mapsto \mathbb{R}$$

Still, this specification of the sine function is not very unique, as many other functions also map real numbers to real numbers. For example, the function $f(x) = 2x$ has the same domain and range. However, not all values of the function range $\mathbb{R}$ are used by the sine function. The sine function maps arguments only to values on the interval $[-1, 1]$. This set of values to which at least argument is mapped is called as the *co-domain* of the function.

$$\sin : \mathbb{R} \mapsto [-1, 1]$$

When the sine function is described by its domain and co-domain, the amount of functions which conform to this specification becomes smaller. The previously shown function $f(x) = 2x$ no longer fits the description, as $f(1) = 2$ which is not on the interval $[-1, 1]$. However, other functions still exist which are consistent with this description. For example, the cosine function has the same domain and co-domain as the sine function.

$$\cos : \mathbb{R} \mapsto [-1, 1]$$

To formalize what is meant with the detail of the function specification, the following definition is used.

Definition: *A function specification s_1 is more detailed than specification s_2 iff the set of all functions conforming to s_1 is a proper subset of the set of functions conforming to s_2.*

In this example, the second description of the sine function is more detailed than the first as the set of all functions conforming to the second is a subset of the first. All functions conforming to the second specification also conform to the first while the opposite is false, as seen by the example of $f(x) = 2x$.

This specification of the sine function using the domain and co-domain is not detailed enough, as similar functions such as the cosine still conform to it. This specification is therefore not useful for software testing. Yet, the sine function maps all values from $\mathbb{R}$ to exactly the interval $[-1, 1]$, which cannot be reduced further. This form of specifying the behavior of the function has reached the limit of the details it can express.

While the specification in the form of the domain, range and co-domain of a function describes the general behavior of the function, it does not describe how elements from the domain are related to elements in the range or co-domain. This difference in mapping is however exactly what distinguishes the sine function from the cosine. To increase the detail of the function description further, the actual relation between arguments and values must be described, which can be described using Hoare logic.

3.4 Hoare logic

This section introduces Hoare logic and shows how it can be used to create a more detailed function specification. Hoare logic forms the basis of the presented modeling formalism.

Charles Hoare developed a formal system to reason about properties of computer programs [Hoa83]. His goal was to provide an axiomatic basis to allow for proofs of correctness. A key element of his formal system is the notion of preconditions and postconditions, which are used to express the properties to be proven. A pair of a precondition and postcondition together describe the relation between the program state before and after execution of a series of instructions. Such properties are expressed as Hoare triplets, which consist of the described instructions in addition to the precondition and postcondition. The notation of a Hoare triplet is as follows.

$$\{ \quad precondition \quad \} \quad program \quad \{ \quad postcondition \quad \}$$

While Hoare developed this formal system to describe properties of instructions during a program execution, Hoare triplets are used to describe properties of functions as well. The notation remains the same.

$$\{ \quad precondition \quad \} \quad function \quad \{ \quad postcondition \quad \}$$

When Hoare triplets are used to describe functions, their semantics are as follows.

$$\forall x \in domain(f) : precondition(x) \implies postcondition(f(x))$$

It is important to note that a precondition does not have to apply to all element in the domain of the function. A Hoare triplet only describes postconditions for arguments that do fulfill the precondition.

Figure 3.2 shows a graphical representation of a Hoare triplet. The arguments that fulfill the precondition from a subset of the domain. The same happens for the values of the range that fulfill the postcondition.

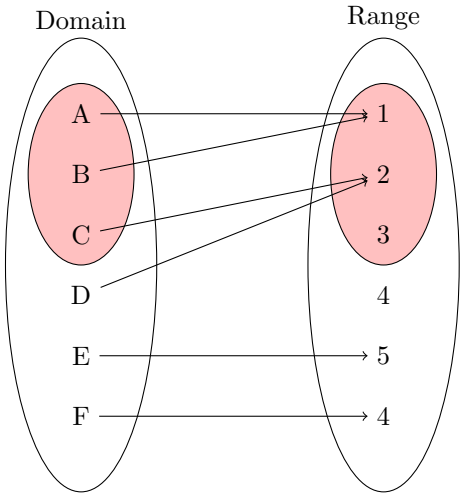

Figure 3.2: Graphical representation of a Hoare triplet. Shown in red, the precondition and postcondition form partitions in the domain and range respectively consisting of all element which fulfill the corresponding condition.

This example already shows the strength of Hoare triplets regarding the detail of function specifications. The function has six arguments which can each be mapped to six different values. This results in $6^6 = 46,656$ different possible functions for this domain and range. When the Hoare triplet is added to the specification, the amount of values to which the arguments A, B and C can be mapped is reduced to three. The other three arguments are not described, and can map to all six values. As a result, the amount of possible functions is reduced to $3^3 \cdot 6^3 = 5,832$.

3.4.1 Strengthening and weakening conditions

An important concept in Hoare logic is the notion of strengthening and weakening conditions. A condition is stronger than another when the set of elements which fit the condition is a proper subset of the other. A condition is weaker than another when the set of elements which fit the condition is a proper superset of the other.

Preconditions can always be strengthened and postconditions can always be weakened [Hoa83, p. 578]. These operations reduce the detail of the specification as either less elements in the domain are described, or they can be mapped to more elements of the range.

On the contrary, strengthening the postcondition and weakening the precondition can not always be done. These operations increase the detail of the specification formed by the Hoare triplet. This is the traditional approach to increasing the detail of Hoare triplet specifications.

When the precondition is weakened, more arguments in the domain are described. Figure 3.3 shows how the Hoare triplet in figure 3.2 can be weakened. The set of arguments which meet the precondition $\{A, B, C, D\}$ is a proper superset of the original set $\{A, B, C\}$. In terms of detail, the amount of possible functions conforming to this specification is reduced further to $3^4 \cdot 6^2 = 2,916$ compared to $3^3 \cdot 6^3 = 5,832$ of the original triplet.

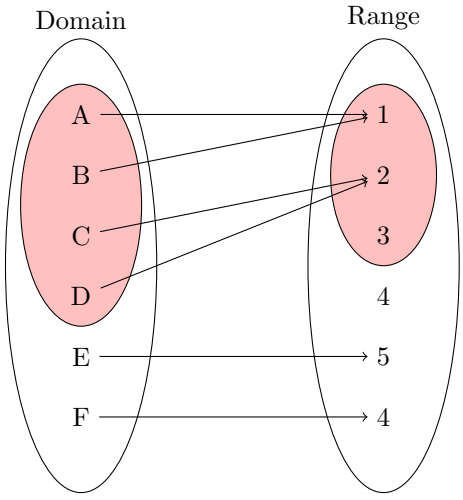

Figure 3.3: Graphical representation of the Hoare triplet in figure 3.2 with a weakened precondition. The weakened preconditions allows the triplets to describe the mapping of more element in the domain.

When the postcondition is strengthened, the mapping of the domain argument is described more concretely by reducing the set of possible values they can be mapped to. Figure 3.4 shows how the Hoare triplet in figure 3.2 can be strengthened. The set of values which meet the precondition $\{1, 2\}$ is a proper subset of the original set $\{1, 2, 3\}$. In terms of detail, the amount of possible functions conforming to this specification is reduced further to $2^3 \cdot 6^3 = 1,728$ compared to $3^3 \cdot 6^3 = 5,832$ of the original triplet.

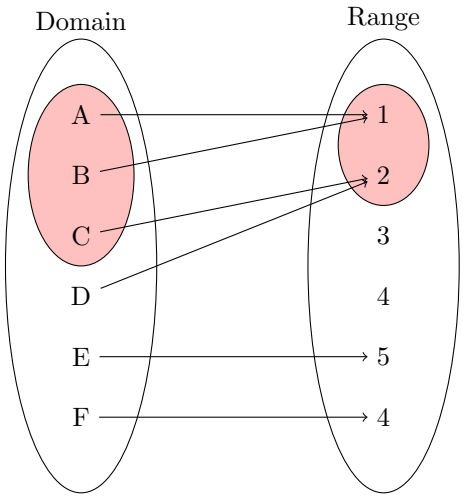

Figure 3.4: Graphical representation of the Hoare triplet in figure 3.2 with a strengthened postcondition. The strengthened postcondition describes the mapping of the elements that fulfill the precondition more closely, as they can be mapped to less values.

Weakening preconditions and strengthening postcondition are the traditional techniques used to increase the detail of Hoare triplet specifications. However, the effectiveness of these techniques is still limited. Figure 3.5 shows the optimal Hoare triplet with both a weakened precondition and strengthened postcondition. The information described by this triplet cannot be increased further. The precondition cannot be weakened further without weakening the postcondition as well. Similarly, the postcondition cannot be strengthened further without strengthening the precondition.

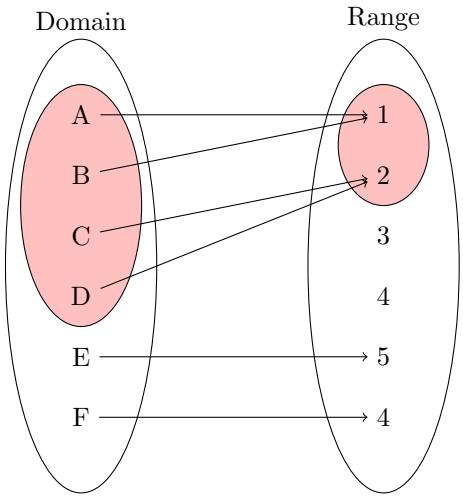

Figure 3.5: Graphical representation of the Hoare triplet in figure 3.2 with both a weakened precondition and a strengthened postcondition.

Luckily, it is still possible to increase the detail of the function specification further. The information described in a single Hoare triplet is limited, so instead of using one, multiple triplets are used. In fact, the original notation of $domain \mapsto range$ can be seen as a Hoare triplet with the weakest possible precondition which was combined with an additional triplet with a stronger precondition. Figure 3.6 shows a possible combination of Hoare triplets to describe the mapping of every argument. To

compare this with previously shown examples, the amount of possible functions conforming to this pair is $2^4 \cdot 2^2 = 2^6 = 64$, which is a massive improvement over the original $6^6 = 46,656$. Such a specification has enough detail to distinguish this function from most similar ones.

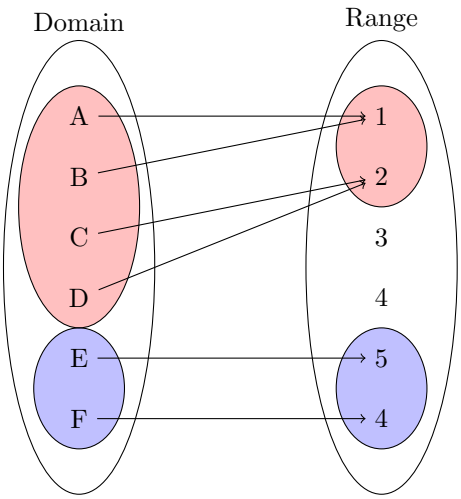

Figure 3.6: Graphical representation of two Hoare triplet, shown in red and blue respectively. The two triplets together describe the mapping of the entire function domain.

Using the standard notation for Hoare triplets, the two triplets in figure 3.6 can be notated as follows.

$$\{ \quad x \leq D \quad \} \quad y \coloneqq f(x) \quad \{ \quad y \leq 2 \quad \}$$
$$\{ \quad x \geq E \quad \} \quad y \coloneqq f(x) \quad \{ \quad y \geq 5 \quad \}$$

This example shows the power of using sets of Hoare triplets to model the input-output behavior of functions. With only two Hoare triplets, the number of possible functions conforming to the specification was reduced massively.

Hoare triplet specifications also allow the trade-off between specification detail and abstraction to be varied according to the problem at hand. When a high level of abstraction is desired, a small set of triplets with weak preconditions can be used. This describes the functions mapping behavior very generally. When a more detailed specification is desired, a larger amount of triplets with stronger preconditions can be used, which allows for stronger postconditions as well. This allows the mapping of arguments to values to be modeled more closely by reducing the amount of possible they can be mapped to.

This combination of expressive power and flexibility in abstraction level make sets of Hoare triplets an excellent formalism to model the input-output behavior of functions and by extension program units for software testing.

3.5 Function modeling with decision tables

This section motivates why decision tables are a good formalism to encode sets of Hoare triplets describing the input-output behavior of functions and program units. It will also show how Hoare triplets relate to decision rules.

3.5.1 Motivation

Section 3.4 motivates why sets of Hoare triplets are an excellent formalism to describe the input-output behavior of functions and program units. However, encoding the Hoare triplets in a decision table offers various improvements to the specification format. This section motivates how the decision table format further improves the usability of the specifications for software testing.

Describing output actions

Hoare triplets express properties of output value when the precondition is met. As discussed in section 3.3.2, program units can have additional output behavior compared to functions in the form of actions with side-effects, such as changing a global or instance variable. These actions do not nicely fit the Hoare triplet notation, limiting their unit-testing functionality to so-called pure functions, programming units which only perform a mapping from arguments to values without side-effects.

Decision tables on the other hand explicitly describe actions performed based on the condition evaluations. Actions with side-effects such as changing a global or instance variable or sending output to `stdout` can be modeled directly as actions in decision tables. To express the postconditions of Hoare triplets in decision table specifications, postconditions to be used as actions as well. These actions can be interpreted as "return a value with this property". This way, decision tables can describe both aspects of program unit behavior.

Automated specification checking

Decision tables separate the conditions and actions from the decision rule logic that relates them. This greatly simplifies the reasoning about properties such as completeness and unambiguity. Describing the behavior of program units with decision table specifications allows the checking algorithms presented in section 3.2.4 to automatically check the specifications. This makes it possible to detect incomplete or ambiguous specifications, which would result in incorrect testing. Separating the conditions from the logic also allows for other operations, such as the optimization algorithms presented in section 3.2.5.

Input partitioning

Most black-box testing strategies, such as the equivalence partitioning and boundary value analysis strategies presented in section 2.1.3, partition the input arguments into sets which elicit similar behavior. The Hoare triplets shown in figure 3.6 allow for easy partitioning, as the two triplets already divide the arguments into two sets relating to distinct output behaviors. However, this is not always the case. Hoare triplets can have overlapping preconditions which apply to the same arguments. This complicates the partitioning as arguments specified by a triplet can be mapped to slightly different behaviors as result of another triplet.

Complete and unambiguous decision tables do have this problem as every possible input transaction matches exactly one decision rule. This creates a clear partitioning on the input domain for which each partition is mapped to a distinct behavior. This allows traditional black-box testing strategies to be used easily for testing decision table specifications.

Compact specifications

Finally, decision tables are a very compact format by extracting possibly redundant conditions and actions and listing them only once. This also provides a clear overview of relevant conditions and behaviors of the program unit. Furthermore, the decision table optimization algorithms also allow the size of the specification to be minimized by removing redundant logic. This improves the readability and maintainability of specifications.

3.5.2 Relation between Hoare triplets and decision rules

As discussed previously, Hoare triplets are implications which describe the output of a function only when the precondition is met.

$$\forall x \in domain(f) : precondition(x) \implies postcondition(f(x))$$

Similarly, as discussed in section 3.2.3, decision rules are implications as well. It might therefore appear that Hoare triplets can be transformed into decision rules one-to-one. However, such a one-to-one conversion would not eliminate the problems of overlapping preconditions. The preconditions of Hoare triplets are allowed to overlap by being applicable to the same arguments whereas decision rules must all be independent. To encode a set of Hoare triplets in a decision table, we must carefully manipulate the triplets so that the resulting decision rules become complete and unambiguous. As an example, observe the following two Hoare triplets.

$$\begin{array}{lllll} \{ & x > 10 & \} \quad y := f(x) & \{ & postcondition_1 \quad \} \\ \{ & x = 20 & \} \quad y := f(x) & \{ & postcondition_2 \quad \} \end{array}$$

When x has the value 20, both triplets are applicable. Because of this, the implications $(x > 10) \implies postcondition_1$ and $(x = 20) \implies postcondition_2$ cannot be used as decision rules. To solve this problem, the triplets are first combined into four new ones describing the combinations of preconditions.

$$\begin{array}{lllll} \{ & (x > 10) \wedge & (x = 20) \quad \} \quad y := f(x) & \{ & postcondition_1 \wedge postcondition_2 \quad \} \\ \{ & (x > 10) \wedge \neg & (x = 20) \quad \} \quad y := f(x) & \{ & postcondition_1 \quad \} \\ \{ & \neg(x > 10) \wedge & (x = 20) \quad \} \quad y := f(x) & \{ & postcondition_2 \quad \} \\ \{ & \neg(x > 10) \wedge \neg & (x = 20) \quad \} \quad y := f(x) & \{ & \quad \} \end{array}$$

Note that a negation of the precondition does not imply that the respective postcondition must fail. Instead, when the precondition does not hold, no properties of the output are specified. Also note that the third triplet with precondition $\neg(x > 10) \wedge (x = 20)$ never holds. It can therefore be removed. The remaining three Hoare triplets are independent, and can never match on the same arguments. These three Hoare triplets can now be mapped to the decision rules shown in table 3.9.

$x > 10$	T	T	F
$x = 20$	T	F	F
$postcondition_1$	X	X	-
$postcondition_2$	X	-	-

Table 3.9: Resulting decision table created from the two previously shown Hoare triplets.

This technique works for any number of Hoare triplets. For n triplets, all 2^n combinations are made and converted to decision rules. The resulting decision rules can then be optimized using the algorithms presented in section 3.2.5.

3.5.3 How this improves usability for software testing

To show how this conversion of Hoare triplets to decision rules improves their usability for software testing, observe the partitions formed by the original Hoare triplets and decision rules in table 3.9. The original preconditions $x > 10$ and $x = 20$ formed two overlapping partitions of the input domain, namely $(10, \infty)$ and $\{20\}$. The partition consisting of all other elements, $(-\infty, 10]$, remained implicit. These partitions are not very suited for software testing. For one, not all elements in the partition $(10, \infty)$ have the same behavior as the case $x = 20$ has an additional postcondition compared to the other cases. Furthermore, these partitions do not reveal the boundaries below and above this case, $x = 19$ and $x = 21$. Finally, the remaining partition $(-\infty, 10]$ must be found by the testing tool itself, and does not have any specified behavior to be tested.

On the other hand, the decision rules shown in table 3.9 form three explicit partitions of the domain with no overlap, $\{20\}$, $(10, 20) \cup (20, \infty)$ and $(-\infty, 10]$. All of these partitions have become explicit, and have a single, consistent behavior. This allows them to be used with traditional black-box testing strategies easily. The fact that no behavior is specified for the partition $(-\infty, 10)$ has also become directly visible. Finally, the boundary values can be taken directly from these domains, which allows these values to be tested as well.

Chapter 4

Implementing a model-based unit testing tool

This chapter shows how the presented decision table modeling formalism is used to develop a model-based unit testing tool. The scope of this tool is unit testing methods and functions written in the imperative programming language Ruby. Ruby is the main programming language used by Axini, which allows the results of the developed testing tool to be evaluated against existing source code and unit tests from their codebase. This experiment is described in chapter 5.

The developed unit testing tool covers the entire model-based testing process. Figure 4.1 shows an overview of the testing tool components and artifacts. Comparing this overview to figure 2.2, an additional constraint solver component is now present. This component is responsible for finding valid transactions for decision rules in given decision table specifications, which is required for checking the specifications properties and for test case generation. Additionally, no adapter component is required for communication between the testing tool and the SUT.

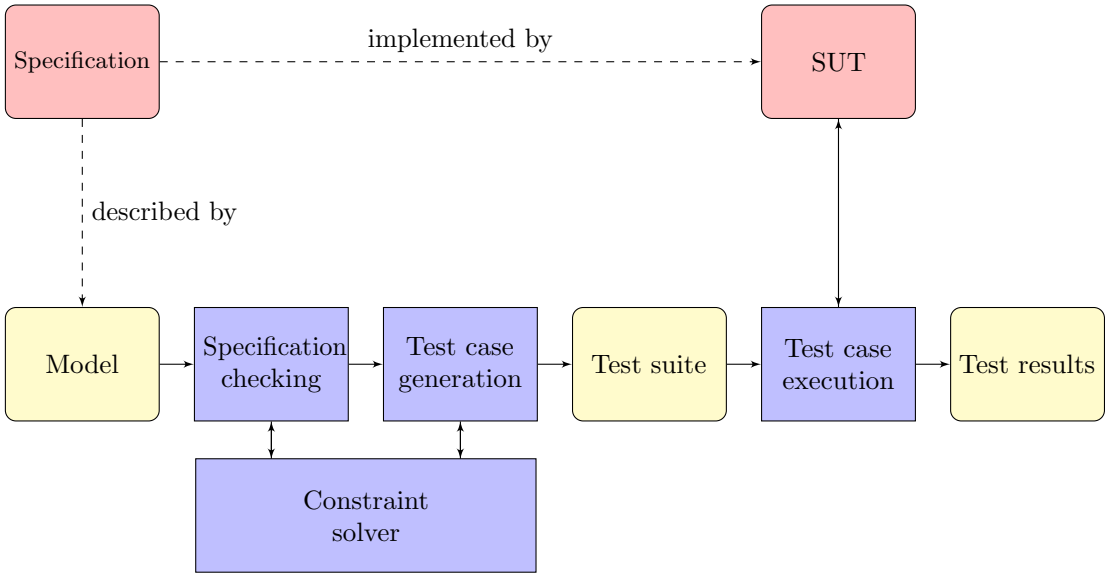

Figure 4.1: Overview of the developed testing tool. Development artifacts are shown in red. Testing artifacts are shown in yellow. The software components in blue are implemented by the testing tool.

The implementation of each of these components is described. The constraint solver component is introduced first, as this component enables specification checking and test case generation. Then, the decision table specification format is presented. The specification format is designed to be compatible with the solver. Then, the developed testing strategy is presented, which dictates the test case selection to generate optimal test suites. At last, test case execution and evaluation is discussed.

4.1 Constraint solver component

4.1.1 Constraint satisfaction problems

Constraint satisfaction problems (CSP) are problems in which variables must be assigned values under certain constraints. Such problems come in many different forms. An example of a CSP is a sudoku puzzle. In a sudoku puzzle, each field in a 9×9 grid must be assigned a value between 1 and 9 in such a way that special rules must hold, namely that each row, column and 3×3 subgrid may not contain the same value twice. A sudoku puzzle is solved by finding an assignment of values to variables (fields) so that none of the constraints on the rows, columns and subgrids are violated.

A constraint satisfaction problem is defined as a triple $< V, D, C >$ where V is a set of variables, D is a set of domains from which the variable can take their values and C is a set of constraints which must hold.

Creating valid transactions for decision rules of given decision table specifications are constraint satisfaction problems as well. A valid transaction is an assignment of values to the transaction variables so that the conditions of the decision table evaluate to the right entry values (`T` or `F`) as specified by the decision rule. In other words, an assignment of values to variables must be found so that none of the conditions evaluate to the wrong entry value. To automate the test case generation for a decision table specification, these CSP must be solved automatically as well.

4.1.2 Constraint solvers

Constraint solvers are programs able to automatically solve constraint satisfaction problems. Different kinds of CSP's require different solving strategies. As a result, constraint solvers are often specialized in a single type of CSP.

In the developed testing tool, GNU Prolog is selected to automate constraint solving. GNU Prolog is an interpreter of the logical programming language Prolog with built-in constraint solving functionality [DC00]. The inherent backtracking capabilities of the Prolog language allow for very efficient execution of the testing strategy presented in section 4.5. Furthermore, GNU Prolog implements the constraint solving strategies much more efficient than competing Prolog constraint solvers [DC00].

GNU Prolog is a *finite domain solver*, which solves CSP's consisting of variables with bounded integer domains and arithmetic constraints over them. To solve constraints on variables with different data types, such as floating point numbers or strings, they must first be mapped to such a finite domain. This will be discussed in more detail in section 4.4.1. The conditions in decision table specifications must be compiled to Prolog predicates to be used as constraints, which is discussed in section 4.4.2.

4.2 Specification checking

As discussed in section 3.2.4, the completeness and unambiguity of decision table specifications can be checked automatically, given that an oracle is able to tell whether valid transactions can exist for given combinations of conditions. Constraint solvers can be used to fulfill this purpose. In that case, the required domain knowledge must be encoded as additional constraints. To check whether a transaction for a combination of conditions can exist, the solver is used to find a transaction for the given conditions in addition to the conditions containing the domain knowledge.

The decision table specification format requires a description of possible values of each transaction variable. This allows the GNU Prolog constraint solving component of the testing tool to automatically check given decision table specifications for completeness and unambiguity. This ensures that the models are valid descriptions of the unit's behavior for testing purposes. This mirrors the model-checking performed on the transition system models by existing MBT tools. In addition to checking the completeness and unambiguity properties of decision table specifications, the testing tool also performs additional checks, such as type checking the conditions.

4.3 Test case generation

The generation of valid transactions for decision rules is closely related to test case generation. A test case consists of both input and the expected output of the SUT. When a valid transaction is found for a decision rule, it can be extended to a full test case by adding the expected output, which consists of the postconditions and actions specified by that decision rule. The problem size of test case generation is quite small compared to other constraint solver applications, such as scheduling. This results in fast execution of test case generation queries, which allows this process to be repeated for different decision rules to build a test suite for the SUT.

4.4 Specification format

The specification format describes a decision table model of the tested program unit to be used by the testing tool. The specification consists of the unit's signature and simple ASCII table containing the conditions, actions and entries. The specification is encoded in a Ruby comment block, marked with `#` symbols. This allows specifications to be embedded in the source files containing the described implementations, which enables the specifications to double as documentation. Figure 4.2 shows an example of a decision table specification placed alongside the tested Ruby function.

```ruby
# describe function sign
# integer x => integer sign
# ------------------
# x  > 0     | T - -
# x == 0     | - T -
# x  < 0     | - - T
# -----------+------
# sign ==  1 | X - -
# sign ==  0 | - X -
# sign == -1 | - - X
# ------------------
def sign(x)
  if x > 0
    return 1
  elsif x == 0
    return 0
  else
    return -1
  end
end
```

Figure 4.2: The decision table specification format is encoded in a comment block to be placed alongside the tested program unit.

The specification format consists of of program unit's signature in addition to the decision table describing the input-output behavior. The signature defines the arguments, their data types, and optionally their domains. These domains can be used as starting point for the finite domain solver, as the variables must have a limited domain of possible values. In the example above, the integer argument `x` is given the default domain $[-1000, 1000]$. The program unit's signature also defines a variable which will represent the output value, in this case the integer variable `sign`. This allows postconditions in the decision table to reference this value.

The conditions and postconditions in decision table specifications use a simple expression language. This expression language is independent of the programming language of the SUT, as it is only used internally for test case generation and test result evaluation. However, the expression language of the testing tool is inspired by the target language Ruby to increases the similarity between specifications and their implementations. This makes the specifications easier to relate to the behavior they model. The data types and operators used in this expression language are discussed in sections 4.4.1 and 4.4.2.

The decision table specification format makes use of mixed entry decision tables. This means that both limited entry rows and extended entry rows are allowed. Figure 4.3 shows an example of an extended entry decision table specification.

```
# describe function sign
# integer x => integer sign
# --------------
# x    | >0 0 <0
# -----+--------
# sign |  1 0 -1
# --------------
def sign(x)
  ...
end
```

Figure 4.3: Example of an extended entry decision table specification.

4.4.1 Data types

The specification format supports the following primitive data types: natural, integer and floating point numbers, booleans, strings, Ruby symbols, immutable strings representing atomic names, and Ruby expressions, strings which will be evaluated as Ruby expressions using the Ruby `eval` function. Ruby expressions allow objects of arbitrary types to be used as argument, even when they are not supported by the specification format, such as Ruby hashes. This comes at the cost of being unable to reason about their properties. Additionally, the specification format supports uniform lists of these primitive data types and (nested) objects with instance variables of these types.

Natural and integer number values can be mapped one-to-one on to finite domains as required by the finite domain solver. The user can specify the domains of such variables in the specification format. Otherwise, default domains are used, as the solver cannot reason about variables with infinite domains. The boolean values *true* and $false$ are mapped to 0 and 1. Other data types, such as floating point numbers and strings do not map directly to finite integer domains. As the solver is only able the reason about variables with bounded integer domains, those data types require a mapping from their values to integers.

Floating point numbers can be mapped to integer domains by reducing their precision to a selected number of decimal d. Then, the values can be mapped to integers by multiplying them by

10^d. This allows the solver to reason about floating point numbers using addition, subtraction and comparison operations. However, other operations like multiplication and division can no longer be used on those numbers, as they prevent the integer values from being scaled back to the correct floating point values.

String and symbol variables have too many possible values to map all of them to integer values. The specification format therefore requires the tester to explicitly list the relevant values of string and symbol variables. This enumeration is used to map the values on a finite domain of equal size. Figure 4.4 shows an example of a decision table specification with such enumerable domains. The integer domains of variables *from* and *to* become $[0, 2]$, which represent the three possible symbol values, which are `:euro`, `:dollar` and `:yen` respectively.

```
# describe function conversionrate
#
# symbol from in { :euro :dollar :yen }
# symbol to   in { :euro :dollar :yen }
#
# => float rate
#
# -------------------------------
# from == to       | T F F F F F F
# from == :euro    | - T T - - - -
# from == :dollar  | - - - T T - -
# from == :yen     | - - - - - T T
# to   == :euro    | - - - T - T -
# to   == :dollar  | - T - - - - T
# to   == :yen     | - - T - T - -
# -----------------+--------------
# rate == 1.0      | T - - - - - -
# rate  > 1.0      | - T T - T - -
# rate  < 1.0      | - - - T - T T
# -------------------------------
def conversionrate(from, to)
  ...
end
```

Figure 4.4: Example of a decision table specification with symbol variables with enumerable input domains.

Lists are represented using Prolog's internal list data type. Prolog can generate lists of given lengths containing unbound variables. Binding these variables to finite domain variables allows them to be solved. An additional variable is introduced to contain the length of the list, which allows conditions to constrain that value as well.

Objects and instance variables

As described in section 3.1, testing methods requires an additional step compared to testing a function. When a method of a class is tested, a new instances of the corresponding class has to be created. The instance variables of that instance can be part of the methods input and output. For this reason, the relevant instance variables of the corresponding class are described in the specification format. These instance variables can then be used in the conditions of the decision table specification to decide how the method behaves. This allows input values for these instance variables to be found so that suitable instances can be created. Instance variables are marked using the `@` symbol, which is the same notation used in Ruby programs.

Instance variable and input arguments can also contain objects themselves. These objects are modeled as containers that contain one or more instance variables of the primitive data types, lists or even nested objects. To reason about such (nested) objects with a finite domain solver, they are first flattened to the set of primitive and list variables. Namespacing is used to ensure all resulting variables are uniquely identifiable. The solver then finds values for these variables similar to normal variables. Afterwards, the found values are used to construct the specified objects bottom up. The objects are allocated without using their constructor, which prevents any computations in the constructor from altering the found values. Instead, the relevant instance variables are set directly using the metaprogramming method `instance_variable_set`. Objects can be of all Ruby classes available in the tested program.

Figure 4.5 shows an example of a decision table specification with an object input of type `Point`. Both the tested `Point` instance and the input `Point` instance contain instance variables `@x` and `@y`, which can be used in the conditions and postcondition of the decision table specification. To create suitable test instances and input instances for this method, the required values of these instance variables are found first. The objects are then constructed with these values.

```
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end

  # describe method direction_to_point of class Point
  #
  # integer @x
  # integer @y
  #
  # Point other <
  #  integer @x
  #  integer @y
  # >
  #
  # => integer angle
  #
  # ---------------------------------------
  # @x <= other.x |   T    F     F     T
  # @y <= other.y |   T    T     F     F
  # --------------+------------------------
  # angle         | >=0   >=90   >=180 >=270
  # angle         | <=90 <=180 <=270   <360
  # ---------------------------------------
  def direction_to_point(other)
    ...
  end
end
```

Figure 4.5: Example of a decision table specification with object input.

4.4.2 Operators

An overview of the implemented operators is shown in table 4.1. These operators can be used to combine transaction variables and literals into conditions and postconditions in the specification format.

data type	operator	description
integers, naturals	`+`	addition
	`-`	subtraction
	`*`	multiplication
	`/`	division
	`%`	division remainder
	`<`	smaller than
	`>`	larger than
	`<=`	smaller or equal to
	`>=`	larger or equal to
	`==`	equal to
	`!=`	unequal to
	`even(Number)`	evaluates whether a number is even
	`odd(Number)`	evaluates whether a number is odd
floats	`+`	addition
	`-`	subtraction
	`<`	smaller than
	`>`	larger than
	`<=`	smaller or equal to
	`>=`	larger or equal to
	`==`	equal to
	`!=`	unequal to
booleans	`&&`	boolean conjunction
	`||`	boolean disjunction
	`!`	boolean negation
	`==`	equal to
	`!=`	unequal to
strings, symbols expressions	`==`	equal to
	`!=`	unequal to
objects	`nil(Object)`	evaluates whether an object is `nil`
	`not_nil(Object)`	evaluates whether an object is not `nil`
lists	`==`	equal to
	`!=`	unequal to
	`length(List)`	evaluates to the length of the list
	`List[Index]`	evaluates to the element at a given index of the list
	`sorted(List)`	evaluates whether the elements of the list are in sorted order
	`not_sorted(List)`	evaluates whether the elements of the list are not in sorted order
	`all_different(List)`	evaluates whether all elements of the list are distinct values
	`not_all_different(List)`	evaluates whether not all elements of the list are distinct values

Table 4.1: Overview of the operators supported by the specification format used to form expressions. As per Prolog standards, placeholder variables are started with an uppercase letter.

To use conditions as constraints in the GNU Prolog solver, they are compiled into Prolog predicates. The arithmetic, relational and boolean operators are all provided by GNU Prolog. Thus, such expression can be compiled easily. More complex operators, such as `sorted` and `all_different` are implemented as additional Prolog predicates using the basic arithmetic operators. For example, the `sorted` predicate uses the `<` operator internally to check whether each pair of subsequent element are in sorted order. These predicates can then be used in the condition predicates. This way, the amount of operations available can be expanded freely. All such additional operators require a negated version, so that conditions can be negated by false (`F`) entries.

In addition to Hoare logic, the specification format can also describe actions taken by the SUT in the form of a state changes and raised errors. The raising of errors and exceptions is modeled using the `raise` keyword along with the expected error or exception. A decision rule can describe whether or not such action is executed using the `X` and `-` entries. Changes to instance variables can be modeled using the `new` and `old` keywords. For example, the postcondition `new @x > old @x` describes that the value of the instance variable `x` is increased by execution of the method. An assignment action can similarly be modeled with the postcondition `new @x == value`.

4.5 Testing strategy

Using a constraint solver such as GNU Prolog, massive amounts of test cases can be generated from a decision table specification. However, an optimal test suite contains a minimum amount of test cases required to fully cover the behaviors of the SUT described in the model. This section describes the developed testing strategy, which dictates which test case are selected into this optimal test suite.

4.5.1 Coverage metrics

The developed testing strategy aims to cover all aspects of the SUT described in the model with a minimum amount of test cases. To do so, a clear definition of test coverage for decision tables specifications is required. Existing model-based testing tools based on transition systems use coverage metrics such as the percentage of states and transitions of the model covered by the test suite. These metrics are closely related to the objects used in the modeling formalism, namely the states and transitions of the transition system. As decision table specifications have a very different structure, different coverage metrics are required.

In white-box software testing, numerous metrics are used to asses how much of the source code is covered by a given test suite. Commonly used coverage metrics are *decision coverage* and *condition coverage*. These two metrics can be adapted to be used for decision table specifications [Mor93, p. 112].

Decision coverage, also known as branch coverage, expresses how many of the possible choices at branches in the control flow are taken at least once. For example, an *if-else* conditional has two branches. When both branches are taken at least once by the test suite, the test suite has a 100% decision coverage. Note that decision coverage is not the same as the percentage of possible control flow paths covered. A code block with two subsequent *if-else* conditionals has four possible control flow paths, yet two test cases are enough to cover each choice of each conditional at least once. In that case, one test case covers both *if* blocks and the other covers both *else* blocks.

Condition coverage expresses how many evaluations of boolean expressions are covered by the test suite. For example, when an *if-else* conditional has the condition `p || q`, the conditional can become true in multiple different situations. The condition coverage of the test suite over the code is the percentage of subexpressions such as p and q and their evaluations (true and false) which are covered at least once. In this example, a test case with p is true and q is false and a test case with q is true and p is false together have a 100% condition coverage, even though they contain only two of the four possible assignments.

Decision tables consist of similar objects. The decision rules represent distinct decisions to be taken, which are chosen based on boolean conditions. As a result, these white-box coverage metrics can be adopted to be used for decision table testing. Decision coverage can be used to express the percentage of decision rules covered by the test suite. Condition coverage can be used to expresses the percentage of condition evaluations (`T` or `F`) covered by the test suite.

These two coverage metrics both have a similar disadvantage. Both metrics can be satisfied with a suboptimal test suite. As discussed above, a complete decision coverage can be obtained with a test suite which does not cover all combinations of decisions. A complete condition coverage can be obtained without testing all possible combinations of boolean subexpression evaluations. The two metrics are used as guidelines for manual white-box testing. Manually designing test cases for all combinations of control flow paths or subexpression evaluations is a tedious and labor intensive task. To limit the manual effort, the metrics describe a minimal level of acceptable coverage of the SUT that covers at least each relevant aspect of the source code once.

However, since model-based testing automates test case generation, it is possible to aim for a better coverage of the SUT. The disadvantage of decision coverage is not relevant to decision tables, as all decision rules are independent. A test case cannot combine multiple decisions. The disadvantage of condition coverage on the other hand is very relevant. Due to immaterial entries in decision table specifications, many condition evaluations are possible which effectively represent the expanded, unoptimized set of decision rules. A minimal test suite generated for condition coverage would exclude many of these relevant test cases, possibly leading to missed execution paths of the tested function or methods.

Instead, the testing strategy aims for a test suite consisting of test cases for all possible combinations of condition evaluations, effectively testing the set of expanded decision rules. This improved condition a coverage is sometimes referred to as *multiple condition coverage*. The developed testing strategy aims to maximize both decision coverage as well as multiple condition coverage of any given decision table specification.

Both the state and transition coverage metrics used for transition systems and the decision and condition coverage metrics used for decision tables are based on the modeled behavior of the SUT. However, a different kind of coverage can be identified as well: *data coverage*. Data coverage expresses how well the domains of the input arguments are covered. As discussed before, full coverage of the input domain is almost always unrealistic. However, carefully selected input arguments, for example using the boundary value analysis strategy, can increase the confidence that the data domain is covered sufficiently well based on the assumption that arguments between the tested values are implemented in a similar way.

4.5.2 Testing priorities

Although the testing strategy aims to generate a minimal test suite, the size of the resulting test suite can still be larger than desired. It is desirable to limit the amount of test cases in the test suite, especially when the execution of a test case is relatively slow. Therefore, the testing tool can reduce the generated test suite further to a size selected by the user by prioritizing certain coverage metrics over others. When the testing strategy is not bounded by the number of test cases, the test suite will maximize all three coverage metrics.

Priority 1: decision coverage

The testing strategy prioritizes a 100% decision coverage, by including at least a single test case for each decision rule. Similar to the equivalence partitioning strategy, this allows the test suite to test whether each distinct behavior of the SUT is implemented.

Priority 2: multiple condition coverage

Next, the testing strategy aims to maximize multiple condition coverage of the decision table specification. The condition evaluations of each decision rule are combined in all possible ways, resulting in all distinct evaluation combinations of that decision rule.

To obtain a full multiple condition coverage, the testing tool must find all evaluation combinations of each decision rule. By default, a decision rule has only one possible evaluation. Disjunctions however increase the number of combinations, as they can have multiple valid evaluations. Disjunctions $p \vee q$ have three evaluations which make them true: either p is true, q is true, or both are true. On the contrary, conjunctions $p \wedge q$ have only one evaluation of subconditions p and q to become true. When a conjunction is negated, for example due to a `F` entry in the decision rule, it is transformed into a disjunction using the laws of De Morgan.

$$\neg(p \wedge q) \iff \neg p \vee \neg q$$

Immaterial entries (`-`) used in decision rules form the primary source of disjunctions. The corresponding condition on that row of the decision table may evaluate to both true and false, which is equivalent to the tautology $(condition \vee \neg condition)$. As the two subexpression are each others negation, they cannot be true at the same time. As a result, each immaterial entry only has two possible evaluations.

Furthermore, to find all evaluation combinations, boolean disjunctions in conditions of the specification must be taken into account as well. These each have three different possible evaluations. The set of all possible condition evaluations is then created by taking the product of the sets of possible evaluations of each immaterial entry and disjunction.

The following example illustrates this process. Observe the left decision table in table 4.2. The first decision rule has both a disjunction in the first condition and an immaterial entry for the second condition. The disjunction in the first condition has three possible evaluations. The immaterial entry has two. Thus, to obtain a full multiple condition coverage over this decision rule, six possible evaluation combinations must be tested. Since the immaterial entry is tested for both $even(p)$ and $\neg even(p)$, the expanded decision table on the right is effectively tested.

$(p > 0) \vee (q > 0)$	T	F	F
$even(p)$	-	T	F
⋮	⋮	⋮	⋮

$(p > 0) \vee (q > 0)$	T	T	F	F
$even(p)$	T	F	T	F
⋮	⋮	⋮	⋮	⋮

Table 4.2: On the left, an example of a decision table with multiple evaluation combinations. On the right, its counterpart with the immaterial entry expanded.

The six resulting evaluation combinations are the following. The second and third decision rule in table 4.2 only have one possible solution. Thus eight test cases are required for a full multiple condition coverage of this decision table specification. Section 4.5.4 presents an efficient search technique to find and iterate over all combinations of subcondition evaluations in a decision table specification.

$$p > 0 \wedge q \leq 0 \wedge even(p)$$
$$p \leq 0 \wedge q > 0 \wedge even(p)$$
$$p > 0 \wedge q > 0 \wedge even(p)$$
$$p > 0 \wedge q \leq 0 \wedge odd(p)$$
$$p \leq 0 \wedge q > 0 \wedge odd(p)$$
$$p > 0 \wedge q > 0 \wedge odd(p)$$

Priority 3: data coverage

The found evaluation combinations represent possible execution paths of the modeled SUT. However, the test cases for these paths can be improved further by applying boundary value analysis to obtain a data-coverage on the specification. For each combination of subcondition evaluations, the solution domains of the transaction variables is calculated. At last, boundary value analysis is applied on each for these solution domains. For n transaction variables, this step adds $2n$ test cases for each evaluation combination, one test case for the upper and lower bound of each solution domain.

Boundary value analysis is only performed on variables with sensible domains, such as natural, integer and floating point numbers. Variables with list, object or enumerable data types do not allow for BVA.

4.5.3 Test case selection

The testing tool can reduce the size of the generated test suite based on these coverage metric priorities. The testing tool divides the amount of available test cases over the decision rules of the decision table specification. First, every decision rule receives a least one test case to ensure a full decision coverage. Then, the remaining number of available test cases is divided over the decision rules proportional to the number of evaluation combinations found in each rule. Test cases for all evaluation combinations are required to maximize the multiple condition coverage of the decision table specification. Finally, the testing tool will select BVA test cases for the test suite only when enough test cases are available to cover all evaluation combinations at least once. In that case, the remaining number of test cases is similarly divided over the decision rules proportional to the amount of available BVA test cases.

4.5.4 Efficient multiple condition coverage using Prolog backtracking

Obtaining a full multiple condition coverage of the decision table specification requires the testing tool to find and iterate over all possible evaluation combinations. For each combination, the constraint solver attempts to generate a valid test case. Invalid combinations with a contradicting conditions do not result in a test case. The resulting test cases then cover all valid subcondition evaluations combinations of the decision table specification.

A naive implementation lists all possible combinations and gives the resulting constraint satisfaction problems to the solver one-by-one. However, such an implementation would be very inefficient, as large parts of the constraint satisfaction problems to be solved would overlap between the different combinations. Furthermore, many combinationss might be invalid for the same conflicting conditions. Luckily, a more efficient searching algorithm can be used to find all valid subcondition evaluation combinations of a decision rule.

The expansion of a decision rule into possible evaluation combinations can be seen as a tree. From the root node down, each condition of the decision rule is subsequently added as a child node of all current leafs of the tree. A path from the root to a leaf then represents an evaluation combination containing all conditions. Every disjunction (both immaterial entries and boolean disjunctions) splits a branch into two or three branches, corresponding to the different possible evaluations of that condition. As a result, each distinct evaluation combination is represented by a branch in the tree. All valid evaluation combinations are represented by branches without any contradicting conditions. This means that when a contradicting condition is found on a branch, the entire branch can be closed as it can no longer result in any test cases.

When the search for valid combinations is performed on such a tree, for example using depth first search, invalid branches of the tree can be closed early to prevent redundant constraint solving. Furthermore, partial solutions shared between multiple branches can be reused.

The following example illustrates this search technique. Table 4.3 shows an extended entry decision table on the left and its expanded limited entry counterpart on the right. For this example, the second decision rule with $x = 0$ and $y = 1$ is used. As the decision rule contains four immaterial entries, $2^4 = 16$ possible evaluation combinations can be created for this decision rule. However, these can only contain one valid combination, as the conditions $x = 0$ and $y = 1$ are mutually exclusive with all other conditions. The immaterial entries can therefore only evaluate to `F`.

Figure 4.6 shows the tree of possible subcondition evaluation combinations of this decision rule. Each immaterial entry splits the tree into two branches while the fixed entries `T` and `F` do not split as the condition has only one possible evaluation. In the figure, red branches contain contradicting conditions. These branches can be closed as soon as a contradicting condition is found. As expected, only one valid combination can be found for this decision rule.

Extended entry table

x	0	0	0	1	> 1
y	0	1	> 1	-	-
⋮	⋮	⋮	⋮	⋮	⋮

Limited entry table

$x = 0$	T	T	T	-	-
$x = 1$	-	-	-	T	-
$x > 1$	-	-	-	-	T
$y = 0$	T	-	-	-	-
$y = 1$	-	T	-	-	-
$y > 1$	-	-	T	-	-
⋮	⋮	⋮	⋮	⋮	⋮

Table 4.3: Example of an extended entry decision table and its limited entry counterpart.

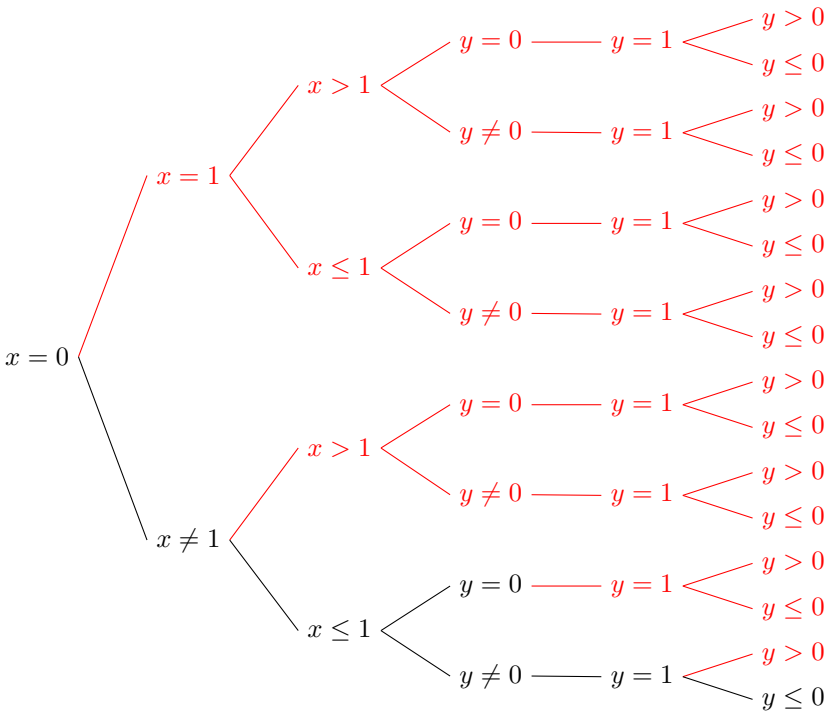

Figure 4.6: The tree corresponding with the possible evaluation combinations of the second decision rule `T - - - T -` of the decision tables in table 4.3. The branches colored in red contain contradicting conditions and can therefore be closed early.

Prolog can be used to implement such a search technique very easily, making use of the internal backtracking functionality of the language. The query to search for all valid branches uses the backtracking disjunction operator `;` on branch splits, which allows the solver to return to the latest branching point when an inconsistent condition is encountered. Moreover, Prolog can use this to iterate over all solutions of the query, resulting in all valid combinations within that decision rule.

4.6 Test case execution

The final step of the testing procedure is to execute the test cases of the generated test suite against the SUT. The testing tool is written in the target language of the SUT, Ruby, which allows the tested classes, methods and functions to be loaded and executed directly. If the testing tool were expanded to more programming languages, an adapter component would be necessary to allow for execution of the tested units.

The expression language of the decision rule's postconditions are evaluated with the selected input values and observed output values of the SUT. Since all variables have a value after the test case execution, no constraint solving is necessary to verify the test result. However, the testing tool reuses the expression compilation to evaluate the postconditions as Prolog queries, which eliminates the need for a dedicated evaluator component for the expression language.

The SUT passes the test when all test cases of the generated test suite elicit a response which complies with the decision table specification. The test results are presented to the user and show how many evaluation combinations were found for each decision rule and how many of the possible test cases were selected into the final test suite.

Chapter 5

Comparing model-based unit testing with manual unit testing

An experiment was performed to evaluate the benefits of model-based unit testing. This experiment involved testing real source code from Axini with the developed testing tool and comparing the generated test suites with the existing unit tests. The tested source code is from Axini's main product TestManager.

Model-based testing promises improved testing quality and reduced testing effort. The experiment attempts to quantify the improvements to test suite quality and reduction of testing effort provided by the testing tool over manual unit testing. Using the testing tool on actual source code also resulted in qualitative insights on the strengths and limitations of the decision table format in practice. Furthermore, two key challenges of model-based unit testing were identified during the experiment. The findings of the experiment are discussed in chapter 6.

5.1 Effect on test quality

Experiment setup

To show how the quality of test suites generated by the testing tool compare to manually written unit tests, the quality of generated test suites is compared against existing unit test found in Axini's codebase. Axini is specialized in software testing. For this reason, it is assumed that the quality of their unit tests is at least up to industry standards.

The developed modeling formalism and testing tool were used to unit test classes and methods of Axini's source code. The tested classes and methods are from Axini's implementation of symbolic transition systems at the heart of their own model-based testing tool. The primary subjects of the experiment are the `Label`, `Transition`, and `Channel` classes, which together represent transitions between states annotated with input and output actions and variable update actions. Some additional methods from other classes were tested as well to add more variety in the data types and operations used in the decision table specifications. An overview of the tested classes in shown in table 5.1.

Axini's source code is accompanied by manually written unit tests developed for the Ruby unit testing framework Rspec. The Rspec framework requires the user to give a description of the tested behavior in each unit test. These descriptions combined with comments and observations of the source code formed the specification used to create decision table models of the methods behaviors. The test suites generated from these models are compared to the manually written Rspec unit tests.

class	methods tested	methods with unit tests	description
Label	15	11	Represents an input or output action label of a symbolic transition system
Channel	4	4	Represents the communication channel of a label, which represents the target of the input/output communication
Transition	8	3	Represents a transition of a symbolic transition system
InstantiatedTransition	3	2	Represents a transition together with its origin state
Variable	3	1	Represent a variable in an symbolic transition system
Valuation	2	2	Wraps around a Ruby hash mapping variables to values

Table 5.1: Classes tested Axini's implementation of symbolic transition systems.

The quality of the manually written and generated test suites is compared based on their coverage over the tested methods. As the testing tool aims for much higher level of coverage than can be expected from a manually written test suite, the test suites are compared using two different coverage metrics.

Combined decision and condition coverage (DC/CC) forms a minimal level of acceptable white-box testing which covers all decisions at branching points and all subcondition evaluations at least once. Both these metrics are used in their original white-box testing definitions, as described in section 4.5.1.

Combined decision and multiple condition coverage (DC/MCC) expands on this by demanding all combinations of subcondition evaluations to be tested. This is a higher level of coverage aimed for by the testing strategy of the developed testing tool.

The amount of distinct test cases required to obtain DC/CC and DC/MCC coverage of the tested methods and the coverage reached by the manually written test suites is determined by manual observation of the source code and test cases. Since the definition of decision and condition coverage for decision table specifications is slightly different from its original white-box context, the coverage reported by the testing tool was verified by hand as well.

Experiment results

As shown in table 5.1, 35 methods are tested for the test suite quality and testing effort experiments. 23 of these methods are accompanied by unit tests to compare with the generated test suites. In 22 of those cases, the testing tool was able to obtain a complete (white-box) combined decision and multiple condition coverage. In 11 cases, the testing tool outperformed the existing unit tests in coverage. In 11 other cases, the test suites obtained the same coverage. In only one case did the testing tool fail to reach a full DC/MCC coverage of the tested method. In that case, the coverage of the testing tool was surpassed by the manually written unit tests. These results are visualized in figure 5.2. Figure 5.1 presents these two coverage metrics of the manually written and generated test suites for the cases in which they differed. The full dataset is available in tables A.1 and A.2 in appendix A.

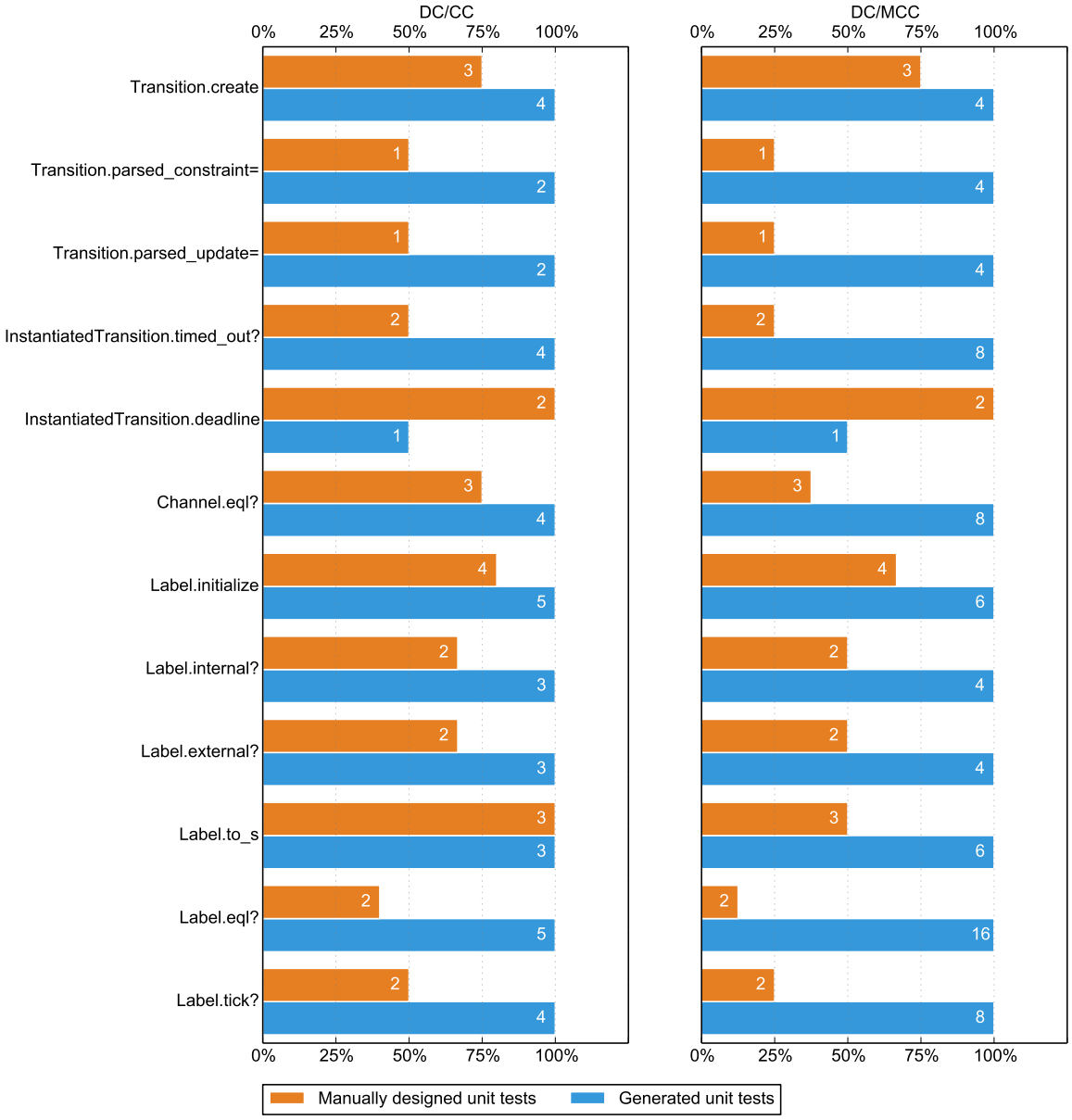

Figure 5.1: Bar plot of the DC/CC and DC/MCC coverage of the manually written unit test suites and the generated unit test suites. The absolute number of distinct paths covered is shown in each bar.

5.2 Effect on required testing effort

Experiment setup

Next, the effect of model-based unit testing on the required testing effort is assessed. As seen before, the generated test suite provides equal or better coverage of the tested methods than the manually designed unit tests with only one exception. This improved coverage is obtained by including more test cases in the test suite. For this reason, it is not sensible to compare the required effort to create these test suites directly. Instead, the required testing effort using the testing tool is assessed in comparison to a fictive, manually developed test suite which obtains the same coverage of these methods with a minimum amount of test cases.

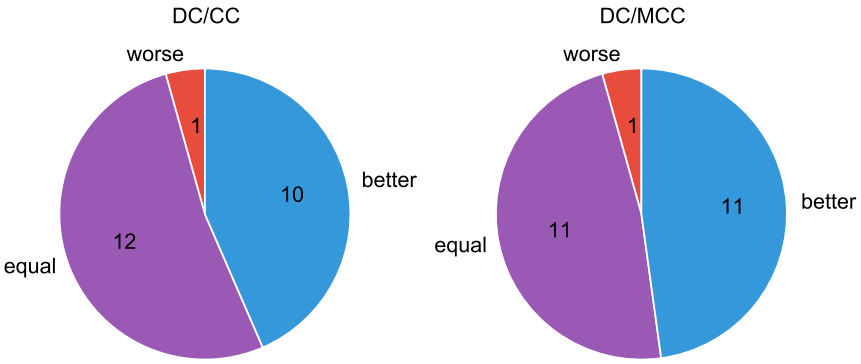

Figure 5.2: Pie plot visualizing how the coverage of the tested methods by the generated test suites compares to the manually designed unit tests.

Manually written test cases are *concrete test cases*, which means that they represent a single input and expected output value of the SUT. The decision rules in the decision table specifications however are *abstract test cases*. Each decision rule represents a set of test cases to test a distinct behavior. The conditions of the decision rule often allow for many different test cases to be generated. Each concrete test case is designed to test a single behavior of the SUT. Such a test case can be considered an abstract test case for which the tester has found suitable input arguments by hand. For that reason, it is assumed that the effort required to create a concrete test case is similar to an abstract test case.

Then, to give an indication of the effect of using the testing tool on testing effort, the amount of abstract test cases (decision rules) required to test Axini's source code is compared to the amount of concrete test cases required to obtain the same coverage. As the amount of required test cases depends on the desired coverage level, the improvement in maintainability is again assessed for both DC/CC and DC/MCC. The minimum amount of concrete test cases required for each tested method to reach these coverages was already determined for the test suite quality assessment. This was done by manual observation of the source code.

For each of the 35 tested methods, the minimum amount of concrete test cases required to obtain a DC/CC and DC/MCC coverage is determined. Then, the amount of decision rules in the decision table specification of that method, after using the optimization algorithms, is taken to be the amount of abstract test cases required to obtain that coverage. The one specifications unable to reach a full DC/MCC coverage of its corresponding method is ignored for this comparison.

Experiment results

The relation between the amount of abstract test cases and the amount of concrete test cases required to test the methods from Axini's codebase is shown in figure 5.3.

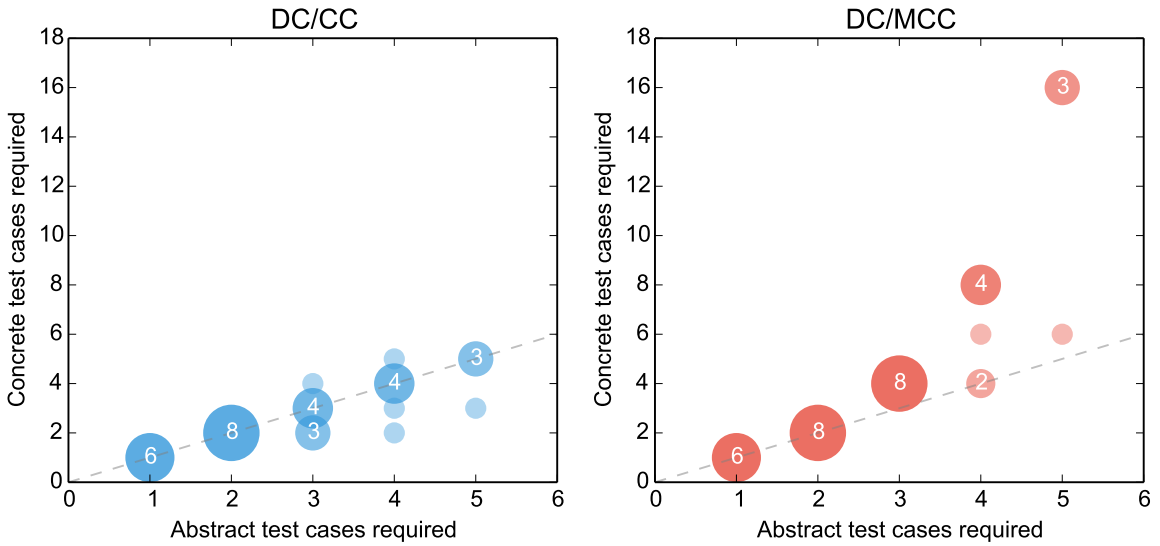

Figure 5.3: Scatter plot of the number of abstract test cases in each decision table specification compared to the number of concrete test cases required to obtain the same complete DC/CC or DC/MCC coverage of the tested method. The numbers represent multiple occurrences of data points. The dashed line $y = x$ shows the break-even point at which the amount of concrete and abstract test cases required to test the method is equal.

Chapter 6

Discussion

This chapter discusses the results of the experiment described in chapter 5 and discusses threats to validity. Furthermore, qualitative insights found while performing the experiment are discussed as well. The strengths and limitations of the decision table formalism and testing tool experienced during the experiment are discussed and two key challenges of model-based unit testing discovered during the experiment are presented.

6.1 Experiment conclusions

Test quality

The results visualized in figure 5.2 show that the generated test suites provides better coverage than the manually designed test suite for around half of the tested methods with reference unit tests. For these 11 methods, the testing tool was able to provide a better DC/MCC coverage than the manually written unit tests. for 10 of those, the tool also provided a better DC/CC coverage. The results in figure 5.1 show that the improvements to both DC/CC and DC/MCC for these methods is in many case quite large as well. While a better DC/MCC coverage was expected, the large improvements to DC/CC coverage is surprising. This means that the manually written unit tests were not able to reach the expected white-box testing standards of complete decision and condition coverage. Reference material on the decision and condition coverage of unit tests in the software industry was not found.

The 11 methods on which the testing tool performed equally well as the manually written unit test all have very low complexity. All of these methods require with one exception at most concrete 2 test cases for a complete DC/MCC coverage. As a result, manually testing all possible paths does not take much effort and thus was done in Axini's unit test suite. For that reason, modeling such methods with low complexity with decision tables did not provide improvements to the test suite's decision or condition coverage. The methods for which the testing tool provided better coverage all require at least 4 concrete test cases for a complete DC/MCC coverage.

The one cases for which the testing tool was unable to reach a complete DC/MCC coverage is related to the type system used in the specification formalism. Ruby is a dynamically typed language, which means that variables can contain objects or values of any data type at any time. The specification format contains a Ruby expression data type to allow input arguments to be dynamically typed. However, the specification format does not have any support for dynamically typed output. This tested method in particular returns output with a dynamic type. It performs a floating point calculation with a possible `nil` output. Since this behavior could not be expressed, the corresponding decision table specification could not be completed. As the Rspec framework specifies all test cases as Ruby input and output, Axini's test suite was able to test for this particular case, obtaining a complete DC/MCC coverage.

To answer **Research question 5a**, complete and unambiguous decision table specifications are shown to obtain a complete DC/CC and DC/MCC coverage over the tested methods. This is also shown to be an improved coverage compared to the manually designed unit tests for all but the trivial to test methods. It can therefore be concluded that the model-based unit testing approach improves testing quality.

Testing effort

The results in figure 5.3 show the relation between the amount of abstract and concrete test cases required to obtain the same DC/CC and DC/MCC coverage of the tested methods. A strong correlation can be seen between the number of abstract and concrete test cases when only a DC/CC coverage is desired. This is not surprising, as the number of distinct behaviors to be described in the model is related to the number of execution paths of the SUT.

On the other hand, the number of concrete test cases required to obtain the same DC/MCC as abstract test cases appears to grow super-linearly. Some data point even suggest an exponential growth. These data points correspond to methods containing a single sequence of boolean logic. Examples of such methods are equality comparisons between objects, which are implemented as conjunctions of instance variable comparisons. Since all combinations of subcondition evaluations are tested, the number of test cases required for a complete DC/MCC coverage becomes quite large. When all n boolean subconditions are part of the same expression, the amount of possible evaluation combinations becomes 2^n.

The reduction in testing effort is only relevant when more advanced coverage such as multiple condition coverage or data coverage is desired, which is uncommon for manual testing. Since the amount of concrete test cases required for DC/CC coverage is similar to the amount of concrete test cases, testing with decision table specifications cannot be said to reduce testing effort. On the other hand, this also means that required testing effort does not increase either.

To answer **Research question 5b**, the testing effort of creating decision table specifications is shown to be similar to manually creating unit tests to obtain a complete DC/CC coverage. While the specification format does reduce testing effort when a better coverage is desired, such coverages are rarely obtained manually in practice. For this reason, the model-based unit testing approach cannot be concluded to reduce testing effort compared to manual unit testing.

On the other hand, to answer the general **Research question 5**, the model-based unit testing approach is shown to improve testing coverage compared to manual unit testing and offers additional benefits such as automatic specification checking without increasing the required testing effort. For this reason, the model-based unit testing approach is concluded to be an improvement over the traditional practice of manual unit testing.

6.2 Threats to validity

The main threat to validity of these findings is the low amount of data points available. Because of this, these conclusions cannot be generalized without further research. The symbolic transition system classes of Axini's codebase together form a complex data structure. Testing methods of these classes required parts of this data structure to be modeled. Modeling these structures was found to be difficult and labor intensive, especially for someone unfamiliar with the codebase. As a result, the experiment was mostly limited to methods which performed computations, boolean logic and object manipulation on smaller parts of these data structures.

Another threat to validity is the quality of the manually created test suite of Axini. Since Axini is a company specialized in software testing, it is not unlikely that the quality of their unit tests is much higher than the average software company. However, the contrary is also possible. Axini aims for a minimum of 98% line coverage, which is relatively high in the software industry. However, this does not ensure that other coverage metrics are also high. No reference material on the decision and condition coverage of unit tests in the software industry was found.

Finally, the modeling formalism might provide more or less benefits depending on the type of behavior tested. Modeling boolean logic in a decision table offers little possibility of abstraction whereas computations can be abstracted massively using Hoare logic. The tested methods from Axini's codebase mostly perform boolean logic and data structure manipulation. A proper evaluation should take this into account by testing a more diverse set of projects and classes.

6.3 Strengths and limitations of the testing tool

The automatic checking of decision table completeness and unambiguity was found to be very helpful in practice. It was found to be especially common to accidentally introduce ambiguity in specifications. The automatic optimization algorithms on the contrary was found to be of little use, as possibilities for immaterial entry optimizations in specifications were often seen easily.

The created decision table specifications often contained similar conditions as the tested source code, which resulted in specifications that are easy to relate to their corresponding implementations. This makes decision table specifications easy to read and check.

The automated reasoning about conditions of the testing tool is implemented using a finite domain solver. As of such, the data types used in the specification must all be mapped to such a finite domain. One common data type used by Ruby that is not supported by the solver is hashes, also known as mappings or dictionaries. Methods using such data type can be tested using the Ruby expression data type with hash expressions such as `"{x: 1}"`. However, this limits automated reasoning and test case generation. A more mature tool should provide a solution for such data types by either implementing a mapping from such datatypes to (lists of) finite domain variables, or by using a more powerful solver.

6.4 Key challenges of model-based unit testing

The experiment also identified two key challenges to be solved before similar testing approaches can fully replace traditional testing frameworks. These challenges are not specific to the decision table formalism, but rather challenges of model-based unit testing in general.

6.4.1 Data structure modeling

The first identified challenge is modeling complex data structures to be used as input arguments or test subjects. An example from Axini's codebase is the symbolic transitions system class. Instances of the STS class contain collections of various connected objects, such as states, transitions, labels, variables and expressions. Moreover, described data structures can require various additional rules on the values and relation between the internal objects. For example, to model a double linked list, the model must describe that each link has a reference to both the previous and next object in the list. As a result, modeling input objects of a function or method operating on such data structures becomes a very daunting task.

Traditional model-based testing tool using transition system based models are not affected by this problem. They can represent the internal state of the SUT using generic datatypes such as lists, sets and hashes. This is possible as the data structures in the model do not have to be identical to the structures in the SUT, as long as the testing tool can reason about their values and properties. Model-based unit testing on the other hand requires such structures to be built so they can be used as input for units or as instances to test methods.

To tests methods of classes with such complex data structures, these structures have to be modeled so that test cases can be generated. Such modeling formalism can be separate from the specification format describing the behavior of the functions and methods. In the field of software architecture and design, structural models are commonly used to describe the components required for the designed system. Examples of such structural models are class diagrams and entity-relationship diagrams. A modeling formalism for data structures to be used for model-based unit testing could take inspiration from these modeling techniques.

6.4.2 Abstractions in modeling formalism

The second identified challenge is using abstractions in specifications. Programming languages use functions and methods to abstract computations which allows them to be reused in other functions and methods easily. The decision table specification format does not support such abstractions. During the experiment, it was found that when a tested method abstracts parts of its logic using another method, the specification would have to contain this abstracted logic as well to reason about suitable input values. This prevents the specification format from scaling up well to methods that use many other methods as abstractions.

Since software often layers such abstractions on top of each other to keep the local complexity of code low, it is essential for a modeling formalism to be able to similarly keep the complexity of the specification low. Otherwise, the specification loses its relation to the code that it describes.

It is possible to build such abstractions with decision tables. To illustrate this, observe the decision table specifications in table 6.1. The left decision table specification of function f makes use of the definition of function g. Since a function g can be any mapping of arguments x and y to a boolean value, a constraint solver would not be able reason about such a black box. However, since the specification of function g is also expressed in a decision table, the internal logic of g is known and can be merged into the specification of function f. The resulting specification of f is show in table 6.2.

describe $f(x, y)$			
$x = y$	F	F	T
$g(x, y)$	T	F	-
result	x	y	0

describe $g(x, y)$				
$x > 0$	T	F	T	F
$y > 0$	F	T	T	F
result	T	T	F	F

Table 6.1: Example of a decision table specifications which reference each other. The decision table specifying function $f(x, y)$ uses the specification of $g(x, y)$.

describe $f(x, y)$					
$x = y$	F	F	F	F	T
$x > 0$	T	F	T	F	-
$y > 0$	F	T	T	F	-
result	x	x	y	y	0

Table 6.2: Example of a merged decision table specification. This decision table specifies the same function $f(x, y)$ as table 6.1, but has the definition of function $g(x, y)$ merged into its own decision rules.

This merging does have some limitations. First of all, the specification of the second function must be complete and unambiguous. Otherwise, the table it is merged into will become incomplete or ambiguous as well. Furthermore, it is only possible to merge the second decision table when the output value of the function is fully specified. This effectively allows the specification to be reversed, so that valid input can be found for a given output value. When the output is only specified using postconditions, such as $result > 10$, the specification can not be reversed, as not all such values might be valid output, nor do all inputs for that decision rule result in the desired value. Such an output can only be merged in when used in a similar condition $g(x, y) > 10$ or a stronger condition $g(x, y) > 20$.

Chapter 7

Related work

This section describes related work on decision tables in the context of software testing and function modeling. This section also describes existing automated unit test generation tools, and discussed how their approach differs from the model-based unit testing approach presented in this thesis.

7.1 Work on decision tables

Decision tables have been used in traditional software testing to help the tester list all the behaviors of the system to be tested. Such approaches are described by Mors [Mor93] and Black [Bla08]. These practices form one of the main inspirations for using decision tables as modeling formalism for model-based unit testing. In this traditional approach, the tester will have to create and select test cases and check the decision table specification by hand. The testing tool presented in this thesis automates these steps. Furthermore, combining decision tables with Hoare logic as shown in this thesis allows the resulting specifications to decsribe properties of computations whereas this traditional approach focuses primarily on actions performed by the SUT.

Modeling functions with decision tables has been studied extensively in the context of model-driven engineering. In model-driven engineering, a formal model is used as a key artifact during the design and implementation phases of software development. A popular research topic was code generation from decision table specifications [Pre65, LT76, Kin66]. This research was concerned with converting decision tables specifications of function behavior into optimal code.

Lamberink has shown the relation between decision tables and symbolic transition systems used in traditional model-based testing [Lam14]. She presented algorithms to convert decision tables into symbolic transition systems and vice versa. This allows decision tables to be used with existing model-based testing tools. However, such an approach limits the effectiveness of using decision tables for model-based unit testing, as the model-checking algorithms and testing strategies provided by these tools are not designed for the resulting models. For example, STS models have no notion of completeness which is an important property of specifications. On the other hand, the properties of STS models which are checked, such as reachability of all states, are not relevant either, as the generated STS models are expected to be correct. The testing tool presented in this thesis implements specification checking algorithms and a testing strategy relevant to decision table models.

Sharma and Chandra have proposed a similar approach to generate test suites from decision table specifications [SC10]. Their tool automatically generates JUnit test scripts from function specifications encoded in simple decision tables. Their tool accepts simple specifications which require little automated reasoning for test case generation. The modeling formalism and testing tool presented in this thesis build further on this concept, by including powerful automated reasoning which allows for more expressive and complex specifications. Furthermore, the decision table format is extended with Hoare logic to model properties of computations in addition to boolean control flow logic. Finally, this thesis also presents a new testing strategy for decision table specifications to select optimal test suites.

7.2 Work on automatic unit test generation

Automatic unit test generation is offered by various tools. These tools can generally be divided into three main approaches.

Tools such as EvoSuite [FA11] and Testful [BM10] automatically generate unit tests directly from source code. The generated test suites form a detailed description of the actual system behavior at the time of generation. Test suites generated by these tool are therefore useful for regression testing, testing whether the behavior of the SUT has changed after refactoring. On the other hand, the generated test suites are not generated from any functional specification of the system. As of such, they cannot be used for functional unit testing.

Tools such as Randoop [PE07] and Jartege [Ori04] do test implementations against functional specifications. These specification are formulated as contracts of methods, inspired by the design-by-contract development approach. These contracts are based on Hoare logic, and each define a single precondition and postcondition of the tested method. The tools then generate random test cases which satisfy the precondition to test whether the postcondition always holds as well. However, as described in section 3.3.3, such approach limits the possible detail of the specification, as the pre- and postcondition must be generic enough to apply to every possible use case. As a result, distinct behaviors of functions and methods cannot be tested. The decision table modeling formalism presented in this thesis allows such contracts to be specified for each distinct behavior of the SUT using Hoare logic, increasing the expressive power of the specifications and thereby enabling more thorough testing.

Similarly, functional program testing tools such as QuickCheck [CH00] and Gast [KATP02] generate random test cases to test properties of functions. Such properties are specified with propositional logic. This allows implication operators to be used to specify conditional properties, which allows different behaviors of the SUT to be specified for different input arguments. However, random test case generation for such conditional behavior does not scale up well to behaviors with complex conditions, as randomly generated test cases have only a small chance to fulfill such conditions. This massively increases the effort required for test case generation. The constraint solving technology used by the testing tool presented in this thesis is able to generate test cases efficiently, even for complex conditions. Furthermore, the decision table format has numerous advantages over sets of such conditional properties, which are described in section 3.5. The decision table formalism can express executed actions as well as properties of computations, enable automatic specification checking, is very compact and relates well to traditional black-box testing strategies.

Chapter 8

Conclusions

Traditional modeling formalism used in model-based testing are not well suited for unit testing. This thesis presents a new modeling formalism for model-based unit testing, which combines the traditional concepts of decision tables and Hoare logic. Model-based unit testing using the presented decision table modeling formalism is shown to be a successful approach to unit testing.

The thesis identifies the behavior of functions and methods which needs to be modeled for model-based unit testing. The modeling formalism should relate input of such units to output behavior. Both the input and output can take various forms. Output behavior can include the execution of actions with side-effects. Instance variables are also identified as possible input and output values of methods.

The thesis also motivates why decision tables combined with Hoare logic form a good formalism to model all this identified behavior. Together they can describe both properties of computations as well as the execution of actions in relation to the units input. The decision rules of a specification also form a clear partitioning of the possible input, which allows such specifications to be used for traditional black-box testing strategies easily. Furthermore, the decision table format allows for automated checking important properties and automatic optimization. Finally, the format is compact, as possible redundant conditions and actions are only listed once.

To show the feasibility of the modeling formalism, a proof-of-concept testing tool is implemented, which performs model-based unit testing using the presented modeling formalism. A description is given of how the testing tool automates all key steps in the model-based testing process. The GNU Prolog constraint solver is used to enable automatic test case generation from decision table specifications. This is done by finding suitable input for each decision rule, and adding the specified output behavior to complete each test case. This solver is also used to implement the algorithms to check the completeness and unambiguity of decision table specification, so that invalid specifications can be detected automatically.

A new testing strategy is presented to select optimal test suites from decision table specifications. This strategy selects a minimum amount of test cases required to obtain complete decision, multiple condition and data coverage over the specification. The size of the generated test suite can be reduced further by prioritizing the coverage metrics in that order. This testing strategy is implemented efficiently using the Prolog backtracking functionality of the constraint solver.

Finally, an experiment has been performed to compare the model-based unit testing approach with manual unit testing. The experiment shows that the developed testing tool obtains better combined decision and condition coverage over methods of Axini's source code than the existing, manually written unit test for all tested methods with exception of methods two execution paths, which are trivial to test. The experiment also shows that the testing effort of creating decision table specifications is similar to manually creating unit test suites to obtain combined decision and condition coverage over the tested methods. This means that the benefits of model-based unit testing do not come at the cost of increased testing effort.

8.1 Future work

The modeling formalism and testing tool could be improved further by experimenting with more powerful automated reasoning, such as satisfiability modulo theories (SMT) solvers. These solvers accept knowledge, called theories, of different data types and structures to reason about them. They can be considered advanced forms of constraint solvers. Examples of SMT solvers are Z3, CVC4 and Alt-Ergo. Using such a solver could allow for solving constraints on more complex data types such as hashes or improve reasoning about strings and floating point numbers.

The experiment comparing the model-based unit testing approach to manual unit testing could be expanded. Further experimentation should involve more source code and unit tests of different projects from different companies. Results of such experiment will be more representative for the software industry as a whole. Furthermore, the effect of the modeling formalism on the test quality and testing effort could be compared for different types of methods, such as computations, boolean logic and data structure manipulation. This could identify situations in which the modeling formalism provides more or less benefits.

Finally, future work can tackle the two identified key challenges of model-based unit testing. The merging of decision table specifications to allow for abstractions in specifications appears to be a promising solution to the first challenge. The feasibility and possibilities of this technique are an interesting topic for further research. The challenge of data structure modeling is also very important, as such modeling formalism is required to create test cases for complex data structures. Future research could try to solve this challenge by taking inspiration from models used in different fields of software engineering or computer science.

Bibliography

[Bla08] Rex Black. *Advanced Software Testing Vol. 1.* Rocky Nook, November 2008.

[BM10] Luciano Baresi and Matteo Miraz. TestFul: Automatic Unit-Test Generation for Java Classes. In *Proceedings of the 32nd ACM/IEEE International Conference on software engineering*, volume 2, pages 281–284, Cape Town, South Africa, May 2010. ACM.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, Montréal, Canada, September 2000. ACM.

[DC00] Daniel Diaz and Philippe Codognet. The GNU Prolog System and its Implementation. In *Proceedings of the 2000 ACM symposium on applied computing*, pages 728–732, Como, Italy, March 2000. ACM.

[FA11] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, Szeged, Hungary, September 2011. ACM.

[Fos80] Kenneth A. Foster. Error Sensitive Test Cases Analysis (ESTCA). *IEEE Transactions on Software Engineering*, SE-6(3):258–264, May 1980.

[Hoa83] Charles Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 26(1):53–56, January 1983.

[KATP02] Pieter Koopman, Artem Alimarinea, Jan Tretmans, and Rinus Plasmeijer. Gast: A Generic Automatic Software Test-system. In *The 14th International Workshop on the Implementation of Functional Languages*, pages 84–100, Madrid, Spain, December 2002. Springer.

[Kin66] P. J. H. King. Conversion of Decision Tables to Computer Programs by Rule Mask Techniques. *Communications of the ACM*, 9(11):796–801, November 1966.

[Lam14] Johanneke T. Lamberink. Using Decision Tables in Model-Based Testing. Master's thesis, University of Amsterdam, May 2014.

[LT76] Art Lew and Doris Tamanaha. Decision Table Programming and Reliability. In *Proceedings of the 2nd international conference on Software engineering*, pages 345–349, San Francisco, USA, October 1976. IEEE Computer Society Press.

[Mor93] N. P. M. Mors. *Beslissingstabellen.* Lansa Publishing, October 1993. Document only available in Dutch.

[MSB04] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* Wiley, Second edition, June 2004.

[Ori04] Catherine Oriat. Jartege: a Tool for Random Generation of Unit Tests for Java Classes. Technical report, Cornell University, December 2004.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 815–816, Montréal, Canada, October 2007. ACM.

[Pre65] Laurence I. Press. Conversion of Decision Tables To Computer Programs. *Communications of the ACM*, 8(6):385–390, June 1965.

[SC10] Mamta Sharma and Subhash Chandra. Automatic Generation of Test Suites from Decision Table - Theory and Implementation. In *Proceedings of the Fifth International Conference on Software Engineering Advances*, pages 459–464, Nice, France, August 2010. ACM.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In *Formals methods and testing*, pages 1–38. Springer, June 2008.

Appendix A

Full experiment results

class	method	DC/CC required	DC/CC manual	DC/CC generated	decision rules
Transition	create	4	3	4	3
Transition	observable?	3	0	3	3
Transition	eql?	7	0	7	7
Transition	without_side_effects?	3	0	3	3
Transition	to_sts?	2	0	2	2
Transition	add_notes	1	0	1	1
Transition	parsed_constraint=	2	1	2	3
Transition	parsed_update=	2	1	2	3
InstantiatedTransition	timed_out?	4	2	4	4
InstantiatedTransition	deadline	2	2	1	1
InstantiatedTransition	eql?	5	0	5	5
Channel	initialize	1	1	1	1
Channel	internal?	2	2	2	2
Channel	eql?	4	3	4	4
Channel	to_s	1	1	1	1
Label	initialize	5	4	5	4
Label	internal?	3	2	3	3
Label	external?	3	2	3	3
Label	unobservable	1	0	1	1
Label	tick	1	1	1	1
Label	to_s	3	3	3	5
Label	eql?	5	2	5	5
Label	stimulus?	2	2	2	2
Label	response?	2	2	2	2
Label	unobservable?	2	2	2	2
Label	observable?	2	2	2	2
Label	tick?	4	2	4	4
Label	inter_model_communication?	2	0	2	3
Label	same_name_channel_parameters?	4	0	4	4
Label	inverse_direction?	2	0	2	4
Variable	initialze	1	0	1	1
Variable	eql?	5	0	5	5
Variable	to_s	2	2	2	2
Valuation	initialize	3	3	3	4
Valuation	[]=	2	2	2	2

Table A.1: Full data obtained by the experiment described in chapter 5 regarding the DC/CC coverage of the manually written and generated test suites.

class	method	DC/MCC required	DC/MCC manual	DC/MCC generated
Transition	create	4	3	4
Transition	observable?	4	0	4
Transition	eql?	64	0	64
Transition	without_side_effects?	4	0	4
Transition	to_sts?	2	0	2
Transition	add_notes	1	0	1
Transition	parsed_constraint=	4	1	4
Transition	parsed_update=	4	1	4
InstantiatedTransition	timed_out?	8	2	8
InstantiatedTransition	deadline	2	2	1
InstantiatedTransition	eql?	16	0	16
Channel	initialize	1	1	1
Channel	internal?	2	2	2
Channel	eql?	8	3	8
Channel	to_s	1	1	1
Label	initialize	6	4	6
Label	internal?	4	2	4
Label	external?	4	2	4
Label	unobservable	1	0	1
Label	tick	1	1	1
Label	to_s	6	3	6
Label	eql?	16	2	16
Label	stimulus?	2	2	2
Label	response?	2	2	2
Label	unobservable?	2	2	2
Label	observable?	2	2	2
Label	tick?	8	2	8
Label	inter_model_communication?	4	0	4
Label	same_name_channel_parameters?	8	0	8
Label	inverse_direction?	4	0	4
Variable	initialze	1	0	1
Variable	eql?	16	0	16
Variable	to_s	2	2	2
Valuation	initialize	4	4	4
Valuation	[]=	2	2	2

Table A.2: Full data obtained by the experiment described in chapter 5 regarding the DC/MCC coverage of the manually written and generated test suites.