Test Sequence Validation and Generation using Classification Trees

Master’s Thesis

Henno Schooljan
Test Sequence Validation and Generation using Classification Trees

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Henno Schooljan
born in Rotterdam, the Netherlands
© 2010-2013 Henno Schooljan. Cover picture taken from [1].
Test Sequence Validation and Generation using Classification Trees

Author: Henno Schooljan
Student id: 1100440
Email: henno@schooljan.nl

Abstract

It has become very tedious to specify test cases for software artifacts in a consistent way, due to high complexity of the system under test. In order to help testers design these test cases more efficiently, tool support has been created in the form of CTE XL Professional to make people able to specify, validate and also generate test cases in a user-friendly way through the use of classification trees.

This thesis project explores techniques for generating and validating sequences of dependent test steps using classification trees, which can only be done manually in the current support tools. We also describe the development process of a prototype plugin for CTE XL Professional for Berner & Mattner Systemtechnik GmbH, which is used to evaluate the test sequence generation process in a practical environment and as a contribution toward inclusion into a future CTE XL release.

We find that exhaustive test sequence generation through the construction of a complete tree of possible test paths is only feasible for very small systems because of time complexity problems, and is only usable for achieving high transition coverage. Simple pseudorandom heuristic test sequence generation methods provide more diverse test sequence sets, however to achieve high test coverage more research is needed on more advanced heuristic algorithms.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. phil. H.-G. Gross, Faculty EEMCS, TU Delft
Company supervisor: Dr. J. Wegener, Berner & Mattner Systemtechnik GmbH
Committee Member: Dr. C. Hauff, Faculty EEMCS, TU Delft
Preface

This thesis report, marking the last step of my MSc studies in Computer Science, is the result of my graduation project at the Berlin office of Berner & Mattner Systemtechnik GmbH, where I did research on test sequence generation using classification trees. It has taken me much longer than anticipated, learning the hard way that combining a job as a web developer and completing a thesis assignment is a very difficult task. I want to thank Gerd Gross for his patience in supervising my project, providing good advice on the structure and contents of my thesis, and for pushing me to complete my thesis even when I did not feel confident in doing so and things did not seem to go forward.

I would also like to express my thanks to Joachim Wegener and Peter Kruse for providing me with the opportunity to do my research at their company in a group of very friendly and supportive people. Next to learning a lot about doing research in a foreign company, my stay in Germany also helped me develop myself on a personal level.

Finally I want to thank my parents, Berend and Nel, for supporting me throughout my educational and professional career and for making it possible for me to do this study.

Henno Schooljan
Rotterdam, the Netherlands
August 2, 2013
# Contents

Preface iii  
Contents v  
List of Figures vii  
List of Tables ix  

## 1 Introduction 1  
1.1 Classification trees ........................................ 1  
1.2 Research questions ........................................ 2  
1.3 Overview of this thesis ..................................... 3  

## 2 Classification Trees 5  
2.1 Systematic test case design .................................. 5  
2.2 The classification tree method ............................... 6  

## 3 Test Case Validation and Generation in CTE XL 11  
3.1 Test case validation using dependency rules ................. 11  
3.2 Test case generation using combination rules ............... 12  
3.3 Test sequences and the classification tree method .......... 13  

## 4 Choosing a Test Sequence Model 15  
4.1 Temporal state models ....................................... 15  
4.2 Measuring quality of temporal state models ................ 16  
4.3 Results ..................................................... 18  

## 5 Adding Test Sequence Validation and Generation 19  
5.1 Background on temporal properties .......................... 19  
5.2 Requirements for validation and generation rules .......... 20  
5.3 Grammar definitions and methodology ....................... 21
List of Figures

2.1 A classification tree and combination table [1] . . . . . . . . . . . . . . . . . . 7
2.2 The main window of CTE XL Professional 2.0.14 . . . . . . . . . . . . . . . . 9
5.1 Evaluation of $\Box(p \rightarrow q)$ using result labeling . . . . . . . . . . . . . . 22
5.2 Tree view of a selected test sequence . . . . . . . . . . . . . . . . . . . . . . 28
5.3 LTL test sequence dependency rule editor . . . . . . . . . . . . . . . . . . . . 29
5.4 Validation of two test sequences . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.5 CTL test sequence dependency rule editor . . . . . . . . . . . . . . . . . . . . 30
5.6 Test sequence generation rule editor . . . . . . . . . . . . . . . . . . . . . . . 31
6.1 Brute force, full coverage, durations with group sizes . . . . . . . . . . . . . . 36
6.2 Brute force, full coverage, durations with sequence lengths . . . . . . . . . . . 36
6.3 Brute force, sequence coverage on 1 second with group sizes . . . . . . . . . . 37
6.4 Brute force, sequence coverage on 1 second with sequence lengths . . . . . . . 37
6.5 Random, sequence coverage on 2 seconds with group sizes . . . . . . . . . . . 38
6.6 Random, sequence coverage on 2 seconds with sequence lengths . . . . . . . . 38
6.7 Random, transition coverage on 0.5 seconds with group sizes . . . . . . . . . . 39
6.8 Random, transition coverage on 0.5 seconds with sequence lengths . . . . . . . 39
A.1 Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
C.1 ATM state model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Brute force, full coverage, durations with group sizes</td>
<td>63</td>
</tr>
<tr>
<td>D.2</td>
<td>Brute force, full coverage, durations with sequence lengths</td>
<td>63</td>
</tr>
<tr>
<td>D.3</td>
<td>Brute force, sequence coverage on 1 second with group sizes</td>
<td>64</td>
</tr>
<tr>
<td>D.4</td>
<td>Brute force, sequence coverage on 1 second with sequence lengths</td>
<td>64</td>
</tr>
<tr>
<td>D.5</td>
<td>Random, sequence coverage on 2 seconds with group sizes</td>
<td>65</td>
</tr>
<tr>
<td>D.6</td>
<td>Random, sequence coverage on 2 seconds with sequence lengths</td>
<td>65</td>
</tr>
<tr>
<td>D.7</td>
<td>Random, transition coverage on 0.5 seconds with group sizes</td>
<td>66</td>
</tr>
<tr>
<td>D.8</td>
<td>Random, transition coverage on 0.5 seconds with sequence lengths</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As we are becoming more reliant on complex software systems in our daily lives, a systematic testing approach is needed to verify the correct operation of these systems. This is especially important when these systems regulate our safety, such as embedded safety logic inside cars or automatic traffic control systems in the transport industry, where the failure of such a system could cause serious harm to people.

While the research domain on software testing is very broad [2], testing software generally involves observing its execution at a certain level under varying conditions such as different input variables, and comparing the result of this execution with the result we are expecting. A challenge is to find and document rare boundary cases that are not very likely to happen, but also need to work correctly every time [3].

Software has become too complex to deal with this manually. Ideally, we execute every line of code under all possible circumstances and compare all results to the expected results. However, this is not feasible because the number of scenarios is simply too big for today’s software systems [4, 5]. This means a subset of these test cases needs to be selected that covers all important test scenarios, and yet still testable within a reasonable amount of time.

This problem becomes even more difficult when sequences of test cases need to be executed. This makes the number of possible testing scenarios grow exponentially, requiring the use of effective rules to limit the amount of valid test sequences while allowing for sufficient test coverage.

1.1 Classification trees

Classification trees are a novel approach for specifying test cases, introduced by Grochtmann and Grimm [6]. CTE XL is a graphical test case design tool initially created by Daimler-Benz Research [1] and is currently being developed further by Berner & Mattner Systemtechnik GmbH [7]. It has been used extensively in industry for the systematic design of combinatorial test cases. Users are able to manually specify a classification tree, and test groups and test cases along with their markings in a combination table, representing a number of test cases or steps to be fed into the system under test. Two notable features of the current development version 2.0 of the tool are the specification of combination rules...
1. Introduction

for the automatic generation of a minimal set of test cases satisfying a combinatorial requirement, and also the specification of dependency rules to verify if test cases conform to certain boolean logic requirements regarding the selected input signals in the test cases.

Next to the specification of test groups and test cases within these groups, it is also possible to specify test sequences and test steps within these sequences. For a test sequence we can specify the measure, representation, type and unit of the input signals. For test steps it is then possible to specify transitions between two test steps which represent the function of how the previous input signal changes to the next input signal (e.g. linear, spline, sine etc.). These transitions can also be given a specific visual style by means of color and line style.

1.1.1 Problem statement

In the current development version of CTE XL, the input and validation of test sequences is rather limited. The user has to input test sequences and their test steps manually, then mark the input signals of these test steps, and provide transition information between the input signals of two subsequent test steps. It is however not possible to automatically generate valid test sequences based on requirements specified by the end user.

1.2 Research questions

Ideally, we would want to make it possible for the end user to provide a number of requirements for test sequences, after which the CTE XL application can automatically generate a complete set of possible test sequences that satisfy these requirements, a process called test sequence generation. Furthermore, it is desired to also be able to verify existing test sequences automatically for validity (test sequence validation). From this we can state the following question:

How can we, using user-specified validation and generation rules, perform automatic validation and generation of test sequences for classification trees?

Because this question is rather broad, we have identified a number of sub questions:

- What rules should the end user be able to specify for test sequence validation and generation?

  This question has to do with the expressiveness of the language that will be used by the end user for determining the set of valid test sequences that may emerge from the generator. However, we have to keep the user-friendliness in mind, by not making these rules too complex and preferably by building upon existing constructs that the current users are already familiar with.

- How can these rules be translated into a complete set of test sequences satisfying these rules?

  Problems that are related to this question are:
1.3. Overview of this thesis

- Can an algorithm be devised that performs this transformation, and what are the limitations of this algorithm with respect to computability and time and space complexity?
- Can we also verify that this algorithm does indeed produce a complete set of test sequences based on the supplied rules?
- How can the end user influence the generation process, e.g. is it completely automatic or can the user steer the process by supplying extra information during the generation phase?

To be able to answer these questions, our goal is to provide one or more new extensions to CTE XL that implement the new test sequence requirement and generation implementation by extending existing extension points in the other plugins of CTE XL. This implementation then serves as a prototype for answering our research questions to test the validity of our hypothesis, and possibly as a basis for an implementation in a future CTE XL release.

1.3 Overview of this thesis

The rest of this thesis report is organized as follows:

Chapter 2 will provide a background on the field of classification trees. Chapter 3 gives a detailed overview of the current implementation of validation and generation rules, acting as a solid starting point for describing our own work. Our choice for a suitable test sequence model is then explained in Chapter 4. Our work on test sequence validation and test sequence generation is detailed in Chapter 5. Chapter 6 then describes the experiments done with our implementation and an evaluation of the results. Finally, Chapter 7 will provide a conclusion in light of our research questions stated in this introduction, list our contributions, and also provides pointers for future work.
Chapter 2

Classification Trees

This chapter provides a brief historical background on the classification tree method, after which the CTE XL Professional tool support for classification trees is described.

2.1 Systematic test case design

Determining good test cases for a black box software test suite is a hard problem, partly because the number of possible test inputs is astronomical for complex software systems. Test case designers aim to design a limited number of test cases to keep the test running time at a doable level, while ensuring that the test coverage is high. A large part of past research on test case design focuses on how test cases can be designed in a systematic way, following a strict set of rules and guidelines [2].

Because it is computationally infeasible to execute all possible test scenarios [4, 5], a choice needs to be made on the subset of test inputs to be tested and which test input categories we have. Partition testing [8] aims to subdivide the input domain into relevant subsets, to be able to choose relevant test cases.

However, a strict set of rules was not specified until the category-partition method was introduced by Ostrand and Balcer [9]. This method is based on partition testing and describes a number of clear steps to follow in order to perform the partition: determining input categories, the input choice for each category, and the constraints between inputs. A test specification can then be generated, and the output of this generator can then be used to refine the choices of inputs and constraints in an iterative fashion.

As pointed out by Grochtmann and Grimm [6], this method still has a number of shortcomings:

- The process starts with a lot of test cases for review.
- There is no good test visualization, making the test specification unclear.
- Extra tools are required to perform at least part of the process.
- There is no strict rule about choosing category type definitions, and creating category hierarchies is not possible.
2. Classification Trees

These properties cause confusion for the tester who cannot see clearly how he/she is supposed to walk through the test case definition process, and it introduces a lot of unnecessary work.

2.2 The classification tree method

In order to classify the input domain of a test object more systematically, Grochtmann and Grimm [6] have introduced the *classification tree method*. It works by dividing the input domain of a test object into *classifications*, which are partitions of the input domain in a specific category. Examples of these are velocity, sample rate or perhaps the color of an object.

Within these classifications we can then define *classes*, which present the different input possibilities of each classification. We identify two important aspects of a set of classes within a classification:

- They represent the complete input domain for that classification. For instance, the class choices for a classification *Switch* could be *on* and *off*, representing all possible values. In case of an unlimited amount of possibilities, it is also common to use input ranges for classes, which is fine as long as all possible input values are covered.

- All classes under the same classification are disjoint. In any given situation, only one class is valid under each classification. In the above example for a switch this is evident because the values are complete opposites, however when having input ranges, the error could be made to describe two overlapping input range classes for car velocity, e.g. 0–50 km/h and 50–100 km/h. If the velocity of 50 km/h is included in both classes, they can both be valid at the same time at that velocity, making the classification tree invalid.

It is also possible to put a classification under a class, representing only that part of the input domain. This *subclass* can then be split into more classes and classifications. While all classes under a given classification will ultimately represent the complete input domain for that classification, defining subclassifications will allow you to better structure the different classes on a semantic level, prevent the tree from becoming too wide, and allow you to collapse the tree to hide certain classes that are irrelevant at that moment.

A good example of this feature is the situation where you define a classification *Color* with a broad set of different colors as classes. When it is important that we distinguish different red shades among these colors, it might be a good idea to put those red shades under a subclassification called *Red* in order to make the tree narrower, deeper, and semantically more well-structured.

Such a tree of classifications and classes is called a *classification tree*. The top half of Figure 2.1 shows such a tree with a number of different classifications and classes. We see classifications partitioning the input domain into small and large objects, into different colors, and into different shapes. Furthermore, we see that a sub-classification is defined to represent different types of triangles.
2.2. The classification tree method

![Classification Tree Diagram](image)

Figure 2.1: A classification tree and combination table [1]

### 2.2.1 Defining test cases using combination tables

Traditionally, test cases were described using text-based representations [9]. Especially when one wants to have a concise overview of different test cases, their lack of good visualization properties makes their usage difficult. In the classification tree method, these test cases can be visualized effectively by use of a combination table. Rows of this table represent test cases, and the columns represent classes in the classification tree. For every test case, the relevant input classes are selected, resulting in a marking of the table.

Figure 2.1 also contains a combination table in the lower half, providing table markings for three test cases. The resulting visualization makes it easy to see the relation between the test cases and the hierarchy of classifications and classes. Exactly one class can be chosen under each classification to form a test case, which in this case amounts to $2 \times 3 \times 5 = 30$ possible test cases. The task of the tester is now to select the test cases that are most likely to expose unwanted behavior, while making sure that all classes are being sufficiently tested (e.g. as a criterion for a minimal number of test cases, it would be wise to use a class at least once in the list of test cases [6]).

Once the combination table has been marked, the test case specification may be transformed to a representation that is suitable for test data generation. After the test data is defined and expected results are derived for every test case, the tests can be performed and evaluated.

A nice aspect of this method is that does not strictly require tool support to perform the
entire process. It is possible to use classification trees for software testing without any aid of a software tool. However, computer support has been introduced to aid the tester with constructing classification trees and test cases by providing an interactive graphical user interface.

In the next section we will introduce CTE XL, a professional software tool for working with classification trees.

### 2.2.2 CTE XL: Current Architecture

Unlike previous versions of CTE XL [10], version 2.0 is based on the Eclipse RCP platform. Using this platform, it is possible to create modular desktop applications using the highly plugin-oriented architecture upon which Eclipse itself is built. This enables developers to utilize all plugins that are available in the Eclipse framework, including common interface elements such as property and outline views, and also data modeling features such as the Eclipse Modeling Framework.

The plugins that constitute CTE XL implement *extension points* to extend the functionality of the Eclipse platform plugins, and in turn provide their own extension points which can then be implemented by developers who want to extend functionality of CTE XL itself. What these extension points actually represent is up to the developer, common examples are the addition of things like file types and menu items. Likewise, the prototype that has been developed as part of this thesis work extends a number of extension points located in several CTE XL plugins.

### 2.2.3 The Editor Window

After launching CTE XL Professional, the editor window is presented to the user. This window is the starting point of any work done with the application, and can hold open multiple CTE data files. Figure 2.2 shows this window. It consists of the following elements:

- The outline view (1) provides quick access to the classification tree structure, while the graphical tree view (2) allows the user to draw the tree and position its elements on a canvas. Furthermore, we see the list of test cases or test steps (3) and the related combination table (4) where classes in the classification tree are marked for every test case or test step. A property view (5) is used to view information related to a selected element.

### 2.2.4 Tags

An important feature in CTE XL is the concept of *tags*. Tags are used to annotate classification tree elements, test cases and test steps with arbitrary metadata. This way the end user can specify extra semantic information for tree elements of test cases that is important for readability, documentation of the execution of the test itself.

### 2.2.5 Test case validation and generation rules

In CTE XL Professional, it is possible to specify rules for the validation and generation of test cases. In Chapter 3 we will examine in more detail how these rules are implemented in
2.2. The classification tree method

Figure 2.2: The main window of CTE XL Professional 2.0.14

the current incarnation of CTE XL, since these rules are a critical foundation on which we have built our own contributions.
Chapter 3

Test Case Validation and Generation in CTE XL

In the previous chapter we have provided a brief background on the CTE XL environment, describing the motivation behind the tool and its global functionality. A critical part of this functionality is the possibility for automatic validation and generation of test cases based on a combination of the modeled classification tree and a number of user-specified validation and generation rules.

This chapter will describe these rules in more detail, focusing on how they are used by the end user and how they are implemented.

3.1 Test case validation using dependency rules

The automatic validation of existing test cases (combination table markings) is one of the more advanced features of CTE XL Professional [7]. It is possible to provide dependency rules in the form of a set of rules, stating which markings of the combination table are valid. This is done using an extendable dependency manager, where the end user can enter dependency rules through one of the plugins that implement the dependency manager interface.

The dependency manager plugin that is currently available implements propositional logic rules for validating test cases. Classifications and classes are logical statements that can either be true or false, and all well-known propositional logic operators such as ¬, ∧, ∨ and → are available for forming new combined statements. By use of a rule checker which can be enabled and disabled at will, every test case will be evaluated and marked as invalid when an active dependency rule produces Boolean false as a result.

Internally, the class and classification names are mapped to their numerical identifiers when the rules are parsed, and it is possible to use these numerical identifiers in the rules instead of the names. This is not recommended for easy identification and rule readability however.

These Boolean rules are very effective for specifying certain disallowed relations between classes and/or classifications within a single test case. Furthermore, as long as the
rule checker is active, these rules are being evaluated after every manual change to the combination table. This effectively acts as a guard for the user entering test case information, signaling that a combination has been selected that is not allowed.

Because the Boolean logic rules only act on separate test cases, it is not possible to validate relations between different test cases using this plugin.

3.2 Test case generation using combination rules

A similar interface is also used for the specification of combination rules. While dependency rules merely tell something about the set of individual test cases that are (not) allowed, combination rules specify the set of test cases that need to be there minimally: the minimal set of test cases. This done by a number of operators that can specify an two-wise, three-wise, minimal or complete combination of classes and/or classifications [7]:

- A complete combination of multiple classes is denoted by the * operator in CTE XL Professional 2.0. For instance, the expression $A * B$ means that every class $A_i$ in $A$ needs to be combined with every class $B_j$ in $B$. If both $A$ and $B$ are classifications with 3 classes each, this would mean that we need at least $A * B = 9$ test cases to meet the requirement of the operator.

- For the minimal combination of multiple classes the + operator is used. This is a more relaxed operator compared to a complete combination, because it is only required to use the involved classes at least once, and not necessarily in combination with all other classes. The aim is to generate a test case set that is as small as possible that satisfies the requirements of this operator, e.g. by using each class only once if possible. As an example, the minimal combination $A + B$ with both classifications having 3 classes can be satisfied with only 3 test cases by combining each of the classes in $A_i$ in $A$ with each of the classes in $B_j$ in $B$.

A complete combination will always satisfy the minimal combination requirement as well, with relatively many redundant test cases however. In the above example 6 out of 9 test cases would be redundant.

- The two-wise and three-wise operators signify that each possible pair or triple combination of involved classes need to be present. When we again look at the example situation, $\text{twowise}(A, B)$ indicates a pair-wise combination of all classes $A_i$ in $A$ and $B_j$ in $B$. We see that this is actually equal to the complete combination of $A$ and $B$. However, this is not the case when we consider more than two arguments, e.g. $\text{twowise}(A, B, C)$. When $C$ also has 3 classes, a complete combination would yield $3 * 3 * 3 = 27$ test cases. A pair-wise combination however only requires that complete combinations of $A$ and $B$, $A$ and $C$, and $B$ and $C$ are present, which can be done with less test cases. The challenge is to minimize the amount of test cases to satisfy the operator requirement.
Armed with this information, it is possible to automatically generate a set of test cases that conforms to these combination rules. Depending on the number of classes and classifications this can result in a lot of test cases, but the user is freed from the burden of entering all test cases by himself and it is guaranteed that the combination conditions are met when the rules are correctly formulated.

In CTE XL Professional 2.0, test cases can be generated in a deterministic fashion by use of Binary Decision Diagrams (BDD). Another promising approach is the use of weights to prioritize certain classes over others, with the idea that certain classes may be more important for the test than others [11].

Test case generation is only possible for a set of individual test cases, combination rules for specifying a complete set of test sequences cannot be entered using the existing CTE XL plugins.

3.3 Test sequences and the classification tree method

From our problem statement we can see that automatic validation and generation of test sequences using the classification method and its CTE XL tool support is not yet possible. This means we need to allow the end user to specify rules that will steer the validation and generation process of test sequences, and devise algorithms for performing the actual generation. An important thing to remember is that our new method of test sequence validation and generation should be as user-friendly and familiar as possible, and should therefore be building on top of the constructs we have introduced in this chapter.

In the next chapter we will provide an argumentation for our choice of the test sequence model that has been implemented, after which the theory behind our implementation is discussed in Chapter 5.
Chapter 4

Choosing a Test Sequence Model

In the previous chapters we discussed the current approach on test case validation using Boolean expressions which the user can enter in the dependency manager of the CTE XL application. By using the rule checker functionality, the end user can quickly see which test cases conform to the active rules, since the validation is also performed while manipulating the combination table. We have also seen how a set of test cases can be generated using combination rules which specify the amount of coverage we need to achieve.

Looking at our original problem statement, it is desirable to extend the current functionality with new constructs. This allows existing end users to easily understand the new rules in the context of their current knowledge, and only the new constructs need to be learned. In our concrete situation, this entails the re-use of the Boolean constructs in the classic dependency rules, and the constructs of the combination rules for test case generation.

This chapter deals with the model choice for such an extension. Our research prior to this thesis provided an indication that temporal descriptions are a promising way of modeling these sequences [12]. However, the requirements from Berner & Mattner also need to be taken into account. In the remainder of this chapter these requirements are discussed, a choice of models is listed, and an argumentation for our choice is given.

4.1 Temporal state models

As pointed out by our previous research [12] there are a number temporal state models that could be suitable for use as CTE test sequence model. We will review these models briefly.

- **Finite state machines** or **finite state automata** [13] are often used for modeling systems that change their state in response to external stimuli, and have a distinct number of states. The transitions between these states can describe extra requirements for these transitions, such as real-time constraints [14, 15] or other conditions. FSMs have an initial state, and possible one or more accepting states that represent the end of system execution.

- **Petri nets** [16, 17] are directed bipartite graphs that describe transitions between states and whether or not these may fire. Because a transition can fire at any time
during the time it is enabled, the execution flow is nondeterministic. Petri nets also allow modeling of parallel systems because multiple transitions can be active simultaneously.

- **Decision trees** [18] are spanning trees starting with a single root node. Each node represents a condition where the different outcomes lead to a different child of the node. This is a simple model to understand, however the modeling of repeating iterations results into duplication of nodes due to the spanning tree requirement.

- **Flow charts** [19] are commonly used to describe the flow of events through different systems such as software, organizations or algorithms. Features of decisions trees can also be modeled by flow charts, this time allowing for convergence to a node from different paths. The semantics of the nodes have been standardized (ISO 5807), of which the condition, decision and transition elements would be minimally needed for specifying test sequences.

- **UML diagrams** [3] can also be used for describing system processes, mostly used for modeling different aspects of object oriented software systems.

During the Berner & Mattner project meetings these models have been discussed, with the goal of choosing a suitable model to build an initial prototype with. Timed variants of decision trees, flow charts and FSMs have been taken into consideration also, to assess the possibility of adding timing constraints to these models.

As a result from these meetings and quality measurements we have chosen to use decision trees for our first prototype, because this model is simple enough for good understandability by both users and developers, and is still expressive enough to be used for test sequence modeling. In the following sections the quality metrics and choice process is explained.

### 4.2 Measuring quality of temporal state models

A small number of possible quality criteria have been mentioned in our previous research [12], namely expressive power, usability, simplifiability, aesthetics and scalability. Here we will give definitions of these criteria and arguments for their importance to different stakeholders. During the Berner & Mattner project meetings more subcriteria were devised for assessing the quality of the different models, which are listed in Appendix A.

- **The expressive power** of a model indicates how many different ideas can be expressed with a model. Because one of the goals of this thesis project is to expand the expressive power of test cases to also include transitions between them, this is an important property of the models we are comparing.

- **Model usability** is another important aspect of a test sequence model. The main reason behind this is that end users have to be able to work with the model in an intuitive way, expanding on the knowledge they already possess about current test
case models in CTE XL. By creating a model that is too complex to understand, the learning curve for these users will be too high and therefore makes the end product unappealing in the market. Next to end users, the developers also need to be able to work with the model in an easy way.

- The **simplifiability** of a model is the ability to extract simpler representations out of a model. This is important for allowing people to concentrate of certain simpler parts of the model without distracting redundant elements. This can also help usability by allowing people to understand parts of the model before trying to understand the model as a whole.

- Models should have good **aesthetics**, which also help to understand and market the model better.

- The **scalability** of a model is the ability to make it larger without excessively harming its computational complexity in time or space. This is a required quality when models become larger over time as the systems that they are based on become more complex.

Each of these quality attributes need to be measured using a defined set of criteria. Appendix A lists the criteria and metrics we have devised during the project meetings, the creation of which will be described here.

### 4.2.1 Metrics

For each quality criterion a number of features have been determined which can be counted. The resulting normalized sum can then be used as a metric for quality comparison. An additional weight attribute has been added to each category to account for its importance to the project.

- For measuring **expressive power** we have counted the possibility for conditions, timing information, loops, subprograms, synchronization of events, parallelism and determinism. These are all features that contribute to the expressive power of a model.

- For **aesthetics** it is important that the model is visually attractive. Factors here are non-overlapping edges, possibility of orthogonal positioning, possibility of a consistent flow direction (e.g. left-to-right or top-to-bottom), the distinct number of symbols used, the amount of connectivity and finally the screen real-estate each model wants to fill up when it gets larger.

- Components for measuring **scalability** deal with the effort to mutate the model, the possible number of paths one can take, and the grow rate of the model when adding a single element.

- For **simplifiability** we accounted for the existence of determinism, the absence of parallelism, the absence of blocking and the number of paths that are possible in the model representation.
4. Choosing a Test Sequence Model

- **Usability** is dependent on human factors and is therefore quite hard to measure without an end product to subject to usability tests. However, the navigability, concept difficulty and how many new concepts are being introduced compared to the current state of affairs are indications on how people respond to the different models.

  Each of these features get a score of 1 (x), 0.5 (/) or 0 (-) depending on how much these apply to the different models.

4.3 Results

From the normalized results in Appendix A we can conclude the following:

- Petri nets and Harel state machines are very expressive, but their complexity makes them hard to read and understand for people who are not familiar with them. Because the amount of added expressiveness is less important than overall usability, this disqualifies these models from consideration.

- While decision trees are not as expressive as the other candidates, they still allow for branching test sequences using conditions, possibly with time constraints, while staying understandable for the average person. This makes them a suitable candidate for a first prototype, also because the complexity of their model is friendly enough for the developer to help get started.

- The two remaining representation types, flow charts and finite state machines, have both decent and comparable scores. They both add loops to the test sequence feature set, which is a required property for concisely modeling test sequences consisting of repeated test steps. Choosing between them is not easy, however flow charts are a bit easier to navigate and understand due to the added concepts being somewhat fewer in numbers and less difficult.

  From these results we can conclude that decision trees are a suitable model to start with, because they provide a good compromise between user and developer understandability and expressiveness.

  In all cases we see that adding the concept of time does not alter the score in a significant way. We have therefore decided to not incorporate timing into our implementation to simplify our first experiments. However, it is feasible that timing constraints could be added later in the form of extra transition properties.

  The next chapters will give details about the implementation of test sequence validation and generation rules, starting with an overview of our language extension to the existing test case validation and generation rules.
Chapter 5

Adding Test Sequence Validation and Generation

We now have described the state of the art in test case validation and generation for classification trees using the current CTE XL toolset, and motivated our choice for a decision tree-based test sequence model. In this chapter we will detail the CTE XL extension that has been constructed to incorporate the dependencies between test cases, which are needed for formulating test sequence constraints.

5.1 Background on temporal properties

Individual test cases represent a certain combination of input at a specific point in time. Test sequences possess additional expressiveness in the form of temporal relationships. One test step in a sequence happens after another test step, which implies a specific ordering of these test steps.

Research on model checking [20] extensively describes rules that can be used for validation of a given order of events. These temporal expressions, also known as temporal logics, provide a constraint on the temporal relationships of any given series of events, just like the Boolean expressions of the classic dependency rules express a constraint on the combination of marked classes within one test case. Popular temporal logics are:

- **LTL (Linear Temporal Logics):** A logic which describes validity conditions for a linear sequence of events. Next to Boolean expressions, it contains a number of temporal operators for relating between different events in time, such as the next event, or any or all events from a specific moment in time. This form of logic is especially well-suited for verifying whether a given set of linear sequences of events conform to a specific temporal condition.

- **CTL (Computation Tree Logic):** This logic models time in a tree structure, where it is not known which future paths will eventually be taken. With LTL, all possible future paths have to conform to the specification, but with CTL this is not a require-
5. Adding Test Sequence Validation and Generation

Instead, ‘for all’, and ‘for some’ quantification constructs are added, to specify for how many branches a temporal condition should hold.

- **CTL*: Unlike CTL, this logic allows all operator to be freely mixed, which makes it more expressive than CTL, but also more complex in terms if implementation and model checking performance.

- **IE-LTL: In principle, classic LTL operates on infinite sequences of events. Bauer and Haslum [21] describe a variant of LTL called IE-LTL, which alter the semantics of LTL where the last element of a sequence is concerned. An example of this is the LTL ‘next’ operator, of which the outcome is undefined in classic LTL when there is no next element. However, when we simply regard a finite sequence as an infinite sequence with the last element repeated endlessly, this problem can be expressed as a classic LTL problem. This can be quite relevant for us when we are dealing with CTE XL test sequences of a finite length.

Vardi [22] provides an extensive comparison between the linear and branching logic paradigms. Grammar-wise, LTL is the simplest temporal logic, since it only adds a small number of extra operators with relatively simple semantics. Because it keeps the propositional logic operators we already have, it would be also at least partly familiar for end users who already have experience with the current CTL XL application. Furthermore, its variant IE-LTL is well-suited for verifying the validity of test sequences as we have them now: a finite linear list of test steps. This makes it, on first sight, the logical choice for the first step in the evolution of test sequence validation in CTE XL. However, we need to make sure a temporal logic is expressive enough for use with test sequence dependency rules.

5.2 Requirements for validation and generation rules

This section describes the requirements we have devised for extending the current CTE XL validation and generation rules. The following section will then detail the implementation of these rules as a language grammar and how the rules operate. The goal here is to answer the first subquestion of our research question:

- What rules should the end user be able to specify for test sequence validation and generation?

5.2.1 Validation rules

A couple main questions that we need to be able to answer with our dependency rules are:

- Which test step can or has to follow another step? (test step ordering).

- What relations between test steps are possible and how are these represented? (test step transitioning).

The temporal operators in a temporal logic dictate the order of events that is allowed, and therefore provide an answer to the question of test step ordering.
5.2.2 Generation rules

With generation rules, one needs to be able to specify a coverage criterion. With individual independent test cases, the maximum coverage possible would be the complete combination of all classes that cover the entire input domain, denoted by the * operator. With sequences however, this set is only a small part of the problem, because it does not say anything about their order or any other interdependence constraint. Bounds need to be set on their length as well, to make sure the operation completes in a finite time and uses a finite memory space. In short, the following questions have come to mind during the project meetings:

- How long can a test sequence be?
  This means there needs to be a definition of minimum and maximum length.

- Which test step configurations can be in the test sequence?
  If there are only a limited number of test step configurations that are allowed in a test sequence, then the number of possible combinations is greatly reduced.

- Do we have constraints on the start and/or end states of a test sequence?
  It can be a quite realistic scenario that a test sequence always ends at the same state or always starts at a specific starting point, e.g. the velocity of a cart test trial. This further limits the amount of possible test sequences.

- How do we measure whether or not we have achieved enough coverage?
  From a reusability standpoint, we are able to specify the allowed set of test steps by evaluating a classic combination rule for test case generation, and use its result as a set of allowed test steps.

5.3 Grammar definitions and methodology

This section describes the definitions for validation and generation rules for test sequences in CTE and provides argumentation for the choice of the used constructs.

5.3.1 Validation rules: LTL and CTL

Appendix B lists the formal LL(k) grammars used for the test sequence dependency rules. This is the grammar type used by ANTLR [23], a Java-based parser generator that can output language recognizers for a number of different languages. Seeing that the existing dependency rules were already being handled by ANTLR and it produces Java code, this was the logical choice for our extended dependency rules. Version 3.2 of ANTLR allows for generation of a lexer and parser, which transforms a textual rule representation into an abstract syntax tree (AST), and a tree parser which will actually traverse the AST to perform the validation itself. Performing the parsing from tokens to AST and the parsing of the AST in separate phases is especially useful when the AST has to be traversed more than once. This prevents the need to re-parse tokens over and over again, which is a more expensive operation than repeatedly traversing the AST.
5. Adding Test Sequence Validation and Generation

Verification by result labeling

The verification of a test sequence using temporal logic expressions is not as straightforward as taking the entire expression and using it to validate each test step in the sequence. When one has operations such as ‘eventually’ and ‘globally’ combined with sub-expressions, the result of these sub-expressions contribute to the result of the higher level expressions. For instance, consider this simple LTL expression:

$$□(p \rightarrow q)$$

This expression indicates that the sub-expression $p \rightarrow q$ must hold everywhere in the timeline (‘globally’ operator). In order to see if this is true, we must first evaluate $p$ and $q$ on a test step, and then evaluate $p \rightarrow q$ based on the results of $p$ and $q$. This procedure has to be repeated for all test steps, and if it holds for all of them, the entire expression holds.

It becomes apparent that such an expression is evaluated in a bottom-up manner: first the simplest sub-expressions are evaluated, which are the Boolean constants $true$ and $false$ and the class and classification names such as $p$ and $q$ above. Their results are combined using Boolean and/or temporal operators to for a new result, and so on and so forth, until we end up with one single result for the entire test sequence.

Looking at the way the AST is traversed by an ANTLR tree parser, we see that it also evaluates the simplest expressions first. Using this property, we can then create a bottom-up formula parser which takes a classification tree representation, a test sequence and an LTL formula, and make it output a positive or negative result. This is done by first labeling all test steps with the results of the simple sub-expressions, and then using these results to label all test steps with new results, and so on. An overview of this labeling algorithm for a specific test step is shown in Figure 5.1.

Completely labeling all test steps with all results from all sub-expressions makes sure we catch every possible dependency between intermediate results, making it a very thorough
approach for dependency rule validation. However, it also adds overhead in case a result for a sub-expression is computed and labeled, after which this label is never used again. Ideally, we would want to determine which part of the test sequence is affected by which sub-expression. Because this is a very complex process by itself due to the exponentially growing number of sub-expressions combined with larger test sequences, it is not clear if this would actually be saving time compared to just labeling all test steps in a naïve manner.

Dependency rule grammar

Next to the Boolean logic operators in the current dependency rules for individual test cases, the LTL dependency rules for test sequences introduce a number of temporal operators which can be used to describe the relation between test steps:

- NEXT $\phi$: For the next test step, condition $\phi$ holds.
- GLOBALLY $\phi$: For this test step and all following test steps, condition $\phi$ holds.
- FINALLY $\phi$: Condition $\phi$ holds in at least one test step from the current test step.
- $\phi$ UNTIL $\psi$: First $\phi$ holds, then $\psi$ holds.

It is possible to combine boolean operators with temporal operators freely, to form complex expressions.

The CTL dependency rules for test sequences also provide two quantification operators or quantors:

- $\forall \phi$: For all paths, condition $\phi$ holds.
- $\exists \phi$: At least one path exists where condition $\phi$ holds.

These rules are able to evaluate a complete set of test sequences for validity instead of just a single test sequence. For example, if you want to test whether or not at least one test sequence eventually reaches the condition where condition $\phi$ holds, the simple rule $\exists$FINALLY$\phi$ is all you need.

Lexers, parsers and tree parsers

Our ANTLR grammars consist of three parts:

- A lexer which accepts the input character stream and outputs a list of tokens
- A parser which accepts the list of tokens and constructs an abstract syntax tree (AST) from it
- A tree parser which accepts the AST and actually performs the validation of test sequences with it.
The benefits of having a separate parser and tree parser is that we can perform multiple passes on the same AST, and easily backtrack on it without having to perform unneeded token parsing operations. It also makes the workflow more clear and it is possible to annotate the AST and use this information during a later operation on the same tree.

5.3.2 Generation rules
Where test sequence validation only deals with checking if a given test sequence conforms to a given dependency rule specification, test sequence generation requires information on the set of allowed test sequences. The main challenge here is to be able to define concise rules that limit the number of test sequences to be generated to a finite value, while it also preferred to have a deterministic generation method that generates the same set of sequences for the same inputs every time it is run.

This section describes our test sequence generation approach. First a definition of our generation rules is given, after which we will describe the algorithms we used for generating the sequences. Finally, the implementation of the relevant parts of the CTE XL plugin is discussed.

Definition of generation rules
A rule for generating test sequences consists of the following parts:

- The algorithm used for generating every sequence.
- The maximum time the generator should be running.
- The coverage criterion type to use.
- The set of valid test steps inside the sequence. A test group with test cases can be specified for this.
- The minimum and maximum number of test steps in the sequence. Combined with the specified test group, this effectively limits the number of test sequences that can be generated.
- An LTL rule which all test sequences need to abide to. This is the most powerful part of the generation rule, because it can limit the number of allowed test sequences significantly.

Coverage criteria We have implemented two test sequence coverage criteria:

- Sequence coverage: This coverage function determines how many test sequences have been generated out of all possible test sequences. This percentage is calculated with $100 \times \left( \frac{T_g}{T_a} \right)$, with $T_g$ being the amount of generated test sequences and $T_a$ the amount of total possible test sequences. Naturally, 100% coverage is reached when all possible sequences have been generated.
5.3. Grammar definitions and methodology

- **Transition coverage:** This coverage function determines how many test step transitions have been generated out of all possible test step transitions. 100% coverage is reached when all possible transitions have been generated. Because a single test sequence can have multiple transitions, this does not require as many test sequences as with sequence coverage.

**Test sequence generation algorithms** Two test sequence generation algorithms have been implemented:

- **Brute Force:** A complete, but naive algorithm is to simply build a tree of all possible test sequence paths. This is expected to be very computationally intensive for all but the smallest test groups and test sequence lengths, but it is relatively easy to implement using a recursive algorithm. We want to know the limits of this algorithm in both computational and space complexity.

- **Pseudorandom generation:** Another generation method is to generate random test sequences that conform to the specification, and then stop the generation process when a certain coverage criterion has been reached. While this could be potentially faster, it is also non-deterministic. Depending on the situation, this can be desired or not.

At first both algorithms build a list of all possible transitions between two test steps based on the set of valid test steps \( \text{testGroup} \) that can be inside the sequence, as shown in Algorithm 1. This list is later used to measure the percentage of transitions that have been generated (transition coverage). Both algorithms then construct one or more trees of test steps which encode all test sequences that have been generated. We do this in a depth-first manner to be able to easily backtrack when an invalid sequence is detected without having to recreate the sequence parts multiple times. Because a tree can only have one root node, we construct a different tree for each possible start step.

**Algorithm 1** Create list of transitions

\[
\text{transitionList} \leftarrow \{\} \\
\text{for } i = 0 \rightarrow \text{length(\text{testGroup})} - 1 \text{ do} \\
\quad \text{for } j = 0 \rightarrow \text{length(\text{testGroup})} - 1 \text{ do} \\
\quad \quad \text{if } \text{testGroup}[i] \neq \text{testGroup}[j] \text{ then} \\
\quad \quad \quad \text{transitionList} \leftarrow \text{transitionList} \leftrightarrow (\text{testGroup}[i], \text{testGroup}[j]) \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\text{end for}
\]

In the brute-force algorithm one or more trees are constructed that contain all possible test sequences. This is done by simply taking the first test step as the root node, and then repeating this process recursively for its children until the maximum tree depth is reached defined by the maximum number of test steps the user has specified. This process is shown in Algorithm 2. Effectively, this becomes a depth-first search where every possible test
sequence is eventually visited. We however do exclude test sequences that do have the same test step twice in a row, because there is no transition between two equal test steps and it would therefore be equivalent to a single test step. Also, we do not always need to generate test sequences shorter than the maximum number of steps because the shorter test sequences are already subsets of these sequences. We do need to do this however when the sequences are validated against a temporal rule or when they have start or end step constraints, because this can put constraints on the start and end part(s) of a sequence.

The pseudo-random algorithm creates test sequences repeatedly by randomly picking the test steps to use while constructing the sequences. After a valid sequence is created, it is added to the worksheet and the process repeats.

### 5.3.3 Revisiting the research question

Now we return to our research question earlier in this chapter:

- What rules should the end user be able to specify for test sequence validation and generation?

In the last sections we have specified requirements for an extension of current CTE XL validation and generation rules. Because the rules have to be easily understandable for existing users and have to describe sequences of test steps in time, we chose to extend the existing Boolean rules with temporal operators. We also added constraints on the length of test sequences and specific start or end steps in the sequences. Together with the grammar specification and algorithms acting on these rules we now have a clear picture on which rules we may use for test sequence validation and generation and how they operate.

### 5.4 Implementation

#### 5.4.1 Tagging and visualization of tree information

The first thing we did was adding decision tree model information to the test steps in a sequence. Because nodes in decision trees have one parent and two children, we added these as parent and children attributes inside of the Attributes tag of every test step. Additionally, we added a condition attribute that models the condition that determines when to take the left path and when to take the right path down the tree. We then made a table-based and graph-based user interface for editing and visualizing a test sequence that is selected by the end user, as shown in Figure 5.2.

#### 5.4.2 Test sequence validation

Figure 5.3 shows the LTL dependency rule editor. By making use of existing extension points in CTE XL, these rules could be added as a new rule type next to the existing test case validation rule editor, while making use of the existing CTE XL interfaces such as sharing the dependency rule list, validating the syntax of a rule, and automatic rule checking.
Algorithm 2 Create exhaustive tree of test sequences

1. \(activeSequence \leftarrow []\)
2. if \(startStep\) is null then
   3. for all \(testStep\) in \(testGroup\) do
      4. generateBruteForce(\(testStep\))
   5. end for
3. else
   4. generateBruteForce(\(startStep\))
4. end if

function generateBruteForce(root)
1. if \(coverageType\) is transition coverage and full coverage is met then
   2. return
3. end if
4. if \(maxTime\) exceeded then
   5. return
7. end if

   push \(root\) onto \(activeSequence\)

   if \(length(\text{activeSequence}) \geq minSteps\) and \(length(\text{activeSequence}) \leq maxSteps\) and \((\text{endStep is null or endStep} == \text{root})\) then
      1. if \(ltlRule\) is null or \(ltlRule\) validates \(activeSequence\) then
         2. insert \(activeSequence\) into CTE XL workspace
            3. add transitions in \(activeSequence\) to coverage data
      4. end if
    6. end if
8. end if

   if \(length(\text{activeSequence}) < maxSteps\) then
      1. for all \(testStep\) in \(testGroup\) do
         2. if \(testStep \neq \text{last(activeSequence)}\) then
            3. generateBruteForce(\(root\))
         4. end if
      6. end for
8. end if
10. pop \(activeSequence\)
end function
Algorithm 3 Create pseudo-random test sequences

repeat
    activeSequence ← []
    sequenceLength ← random(minSteps,maxSteps)

    for $i = 1 \rightarrow sequenceLength$ do
        repeat
            step = random step from testGroup
            if startStep is not null and $i == 1$ and $step \neq startStep$ then
                continue
            end if
            if endStep is not null and $i == sequenceLength$ and $step \neq endStep$ then
                continue
            end if
            until activeSequence is empty or step \neq last(activeSequence)
            push root onto activeSequence
        end for

    insert activeSequence into CTE XL workspace
    add transitions in activeSequence to coverage data
until maxTime exceeded or coverage criteria met

Figure 5.2: Tree view of a selected test sequence
5.4. Implementation

We see the extra operators in the form of extra buttons, and the familiar editor area. Upon validation by the rule checker, the `check()` validation function of the rule’s dependency manager is ran for every test step in a sequence, just like it is being ran for every test case with the classic dependency rules. For test sequence validation however, we only want to check the first test step in the sequence because we need to save information about the relation between multiple test steps using the labeling method described earlier in this section. Therefore, the rule checks on all test steps other than the first one can simply return the result from the labeling method without having to compute anything. The entire test sequence will then be coded green or red depending on the outcome of the validation function, as shown in Figure 5.4.

Likewise, Figure 5.5 shows the CTL dependency rule editor, containing the extra CTL constructs.

5.4.3 Test sequence generation

The test sequence generator interface is shown in Figure 5.6, which builds upon the extension points of the generator rule manager. This manifests itself as an extra tab next to the BDD and priority based test case generator rules that already existed in CTE XL. In this window, we can specify the algorithm, run time, coverage criteria, list of valid test steps and other components of our test sequence generator rule, along with two buttons to generate the test sequences, or benchmark the generation process.

The Generate button will generate the test sequences based on the rule that has been set. The Evaluate button will start a benchmark measuring the time it takes to complete the
5. Adding Test Sequence Validation and Generation

Figure 5.4: Validation of two test sequences

Figure 5.5: CTL test sequence dependency rule editor
5.4. Implementation

5.4.4 Revisiting the research question

We can now answer part of our other research questions:

- Can an algorithm be devised that performs this transformation, and what are the limitations of this algorithm with respect to computability and time and space complexity?

A brute-force and pseudorandom test sequence generation algorithm has been devised, however we still need to measure their performance using a case study. This will be done in the next chapter.
5. Adding Test Sequence Validation and Generation

- How can the end user influence the generation process, e.g. is it completely automatic or can the user steer the process by supplying extra information during the generation phase?

As soon as the generation process is started it completes automatically. The user can however supply different parameters to influence the generation process in the test sequence generator interface that we just discussed in this section.

In the next chapter we will describe the benchmarking experiments we have done on test sequence generation and evaluate the results.
Chapter 6

Evaluation

This chapter describes the measurements we have done for assessing test sequence generation performance, and the results from these measurements. First the environment in which the experiments are conducted is described, after which we will evaluate these results.

6.1 Environment

For the experiments, we provide a number of example classification trees in the CTE XL editor, along with a set of test sequence generation rules. The goal is to assess how well our algorithms perform given the same situation, and how different changes to the rules affect the performance.

For the model, we used a simple implementation of an ATM machine, with its different states encoded as valid states in the classification tree. These states are described below and its FSM specification is given in Appendix C.

6.1.1 Model states

These states represent the different screen outputs the machine can have.

- **AskCard**: The machine is asking for a card.
- **ProcessingCard**: The machine has received a card, and is currently busy processing it.
- **InvalidCard**: The machine has rejected an inserted card.
- **EnterPIN**: The machine is asking for PIN input.
- **InvalidPIN**: The machine has rejected a PIN input.
- **ChooseAmount**: The machine is asking for the amount of money to be withdrawn.
- **ProcessingWithdrawal**: The machine is processing the money withdrawal.
- **WithdrawalFailed**: The machine reports that the withdrawal has failed.
• **WithdrawalSucceeded**: The machine reports that the withdrawal has succeeded.

• **End**: The machine greets the customer goodbye.

There are also a number of states that represent the status of other components in the machine, one of which is the card status:

• **CardInserted**: There is a card present in the machine.

• **CardNotInserted**: There is no card present in the machine.

The PIN status also has two states:

• **PINConfirmed**: A valid PIN has been entered.

• **PINNotConfirmed**: A valid PIN has not yet been entered.

Finally, the status of the money is taken into account:

• **MoneyTaken**: The user has taken the money from the machine.

• **MoneyNotTaken**: The user has not yet taken the money from the machine.

From this it is easily seen that these four sets of states are in fact classifications in the tree, and that there are

\[10 \times 2 \times 2 \times 2 = 80\]

possible markings of this tree. The combination

\((\text{AskCard, CardNotInserted, PINNotConfirmed, MoneyNotTaken})\)

represents the starting state of the machine.

Using test case validation rules it is possible to define the valid and invalid combinations, however this functionality was already present in CTE XL. For our own work we are interested in the valid transitions between these states. For example, it would be strange to go from the starting state to a state where money can be taken without ever confirming the PIN number, even if the individual states all check out.

### 6.2 Generating the test sequences

We try to generate test sequences by fixing all but one variable, and changing one variable throughout the test. These variables are:

• **Size of set of valid test steps** (groupSize).

• **Minimum number of test steps in sequence** (minSteps).

• **Maximum number of test steps in sequence** (maxSteps).
6.3 Discussion of results

- Whether or not we have a fixed start step (startStep) and/or end step (endStep).
- The maximum amount of time the test will run.
- The coverage type to measure (All: sequence coverage, Transition Pairs: transition coverage).

During initial tests we have noticed that the actual insertion of generated test sequences in the CTE XL user interface took a significant amount of time when there are many test sequences already in memory. Because this time is not part of the actual generation phase, we do not want these delays to skew our results. Therefore, we left the UI display of results out of the test runs when the Evaluate button is pressed in Figure 5.6. Of course the sequences will be inserted when the Generate button is pressed, but without any benchmarking of the generation process.

For these benchmarks we used an Intel Core2 Duo desktop computer with CTE XL Professional 2.0.14.

6.3 Discussion of results

The complete benchmark data is included in Appendix D. In this section we will describe each experiment in more detail and evaluate their results in light of our research questions.

6.3.1 Exhaustive search with varying sequence length and groupSize

An interesting test is to measure how long it takes to generate all possible test sequences made out of a finite set of distinct test steps, e.g. a set of test cases generated by a classic or prioritized combination rule. It stands to reason that not many test sequences can be constructed out of 2 distinct test steps (2 sequences would be possible), however with an increasing number of possible test steps the amount of possible test sequence combinations becomes large as well. This test therefore measures the scalability of test sequence generation with large sets of test steps to choose from. The results are visualized in Figure 6.1 and Table D.1.

We see that for small sets of valid test steps full coverage is reached relatively fast, but the duration is climbing in an exponential manner. From 20 steps it becomes rapidly infeasible to compute a complete set of test sequences. This may suggest we need to come up with a better algorithm, or accept that we can not get a complete set of test sequences and have to settle with the sequences that are covering a part of the domain.

We repeated this test with varying test sequence lengths, shown in Figure 6.2 and Table D.2. This shows an even more dramatic result, showing a huge increase in duration once a ninth step is added to the maximum sequence length.

These results indicate that a naive brute force approach for test sequence generation scales very badly, it takes far too long for any realistic scenario.
6. Evaluation

6.3.2 Exhaustive search with limited run time

Even if we do not achieve full coverage, a high partial coverage is still a good result compared to manually entered test sequences. Therefore, we also tried the brute force algorithm using a limited run time of 1 second per run, to see how many of all possible test sequences will be generated when again varying the set of test steps and the sequence length.

Figures 6.3 and 6.4 and Tables D.3 and D.4 detail the results. Figure 6.3 measures the percentage of possible test sequences that we generated (sequence coverage) against the amount of test steps we can choose from (group size) when running the brute force algorithm for 1 second. We see that the relative amount of generated test sequences drops quickly when the group size reaches 15. In Figure 6.4 we measured the maximum length of the test sequences against the sequence coverage. Here we see that we cannot generate
6.3. Discussion of results

Figure 6.3: Brute force, sequence coverage on 1 second with group sizes

Figure 6.4: Brute force, sequence coverage on 1 second with sequence lengths

many test sequences in one second when the sequences get longer than around 8 test steps. This again shows that there is a steep drop of effectiveness when the amount of possible test steps or the possible length of test sequences become larger. This confirms that for very complex and large problems, we can not achieve great coverage results quickly using a brute force approach.

6.3.3 Random search with limited run time

Figure 6.5 and 6.6 and Tables D.5 and D.6 show the results when using the strategy of random creation of test sequences.

In the experiment with varying the amount of possible test steps as displayed in Figure 6.5, the results are even much worse than with the brute force strategy. We again compared the sequence coverage to the group sizes and sequence lengths like we already showed in Figures 6.3 and 6.4, but this time we used the random algorithm and doubled the execution time to 2 seconds. Even when actually doubling the run time, we see that the drop in sequence coverage still happens at smaller group sizes and sequence lengths.
6. Evaluation

This was a surprising result, as we initially hypothesized that a random choice of test steps would diversify the results enough from the start to actually generate many different test sequences. Instead, relatively few test sequences emerge from this strategy.

A possible explanation when looking at the algorithm itself, is that there is no deterministic mechanism that remembers which sequences have been generated in previous runs, and that therefore much time is wasted by creating test sequences that already have been generated.

Sequence vs. transition coverage

When looking at the same experiment with varying sequence lengths, we see that we also perform worse compared to the brute force strategy. Clearly, both the brute-force and the random heuristic strategies for generation of test sequences are not very suited to achieve test sequence coverage. However, this can also mean we are measuring our coverage
6.3. Discussion of results

Figure 6.7: Random, transition coverage on 0.5 seconds with group sizes

Figure 6.8: Random, transition coverage on 0.5 seconds with sequence lengths

wrong way. Note again that test sequence coverage is defined as the number of generated test sequences vs. the total amount of possible test sequences. These test sequences will overlap each other on many occasions, which seems a bit redundant when you only care about having many different transitions between test steps.

Therefore we also tested how fast we can achieve transition coverage, which could be reached faster because there are $n$ (possible distinct) transitions in a test sequence of length $n + 1$. Our hypothesis here is that a good coverage ratio can be reached in a much shorter time, even when the sequence lengths or set of possible test steps become large.

6.3.4 Transition coverage with limited run time

The results for transition coverage using the random heuristic are visualized in Figures 6.7 and 6.7 and Tables D.8 and D.8.
When keeping the test sequences small, we can get good coverage for large group sizes, and vice versa. Furthermore, from Figure 6.7 we can see clearly that transition coverage does not change significantly when we keep the amount of possible test steps constant.

6.3.5 Revisiting our research questions

Remember these subquestions of our second research question:

- Can an algorithm be devised that performs this transformation, and what are the limitations of this algorithm with respect to computability and time and space complexity?

We have seen that there are serious time complexity limitations with our algorithms as soon as the number of distinct test steps or the test sequence length become larger. Realistic classification tree models are already much bigger than our test model, which means that test sequence generation will become an extremely time-consuming task. We have also seen that RAM usage can be high when using Java object pointers in our model, this was however easily mitigated by using integer identifiers instead. Time complexity is clearly the bottleneck at this moment.

- Can we also verify that this algorithm does indeed produce a complete set of test sequences based on the supplied rules?

If we disregard the amount of time it takes, the brute force algorithm will eventually generate a complete set of test sequences because it visits all test step combinations in a deterministic way. With time constraints however this is not the case for all but the smallest problems, which is bad for real-life application of this algorithm. Pseudorandom heuristic based generation of test sequences does not produce a complete set in a reasonable time frame as well, however it will produce a more diverse range of test sequences which can be used to test different parts of a software system more quickly.

This indicates that we might have to accept that getting a complete set of test sequences is not realistic for real-life problems, and that more research into better heuristic test sequence generation methods can be very useful in order to increase test sequence coverage.

6.4 Threats to validity

In our experiments we systematically measured what happens to the test sequence coverage figures when we change the test sequence length or the size of the pool of test steps to choose from. From this we can see that the coverage figures already drop very fast with a relatively small classification tree model. We also did not change more than one variable during an experiment. The possibility of confounding is however still unclear when also selecting a fixed starting or end step, or when applying an extra LTL rule. Due to time constraints these scenarios have not been visited, which could be an interesting step towards future work.

An important factor is the machine on which the experiments are run. We used an Intel Core2 Duo powered computer for taking the measurements shown in this chapter, and it is
expected that the results will differ on a computer that has lower or higher computing power. Because there are many variations of models and settings, this can be a hurdle in recreating these experiments for comparison and/or validation purposes. The same argument can be given for different CTE XL releases, because of performance differences in the CTE XL core.

6.5 Lessons learned and open issues

We have seen that is its very hard to achieve high coverage with test sequence generation on bigger classification tree models using simple exhaustive or heuristic algorithms. It would be very interesting to see how a more advanced heuristic algorithm such as hill-climbing or genetic algorithms would perform regarding time complexity. Due to a limited implementation time we did not have a chance of doing this, and this would therefore be a good future research opportunity. It is especially important to eliminate invalid test sequences as early in the process as possible, because now a lot of computing time is lost by generating complete test sequences that are rejected at the end of the algorithm.

Because of the performance issues during our experiments, a complete validation of our ATM model also showed to be infeasible, which is the reason this is not included in our benchmark results. It simply took too long to completely generate the complete set of valid test sequences due to the time it takes to LTL validate each generated test sequence. Because the current CTE XL rule checker actually works on each individual test step instead of each test sequence, there might be room for performance improvement by making the CTE XL rule checker test sequence aware.

Another interesting area that could still be explored is a long experiment run time on big projects. In order to get results in a timely fashion we could not do this, but this could shed more light on the feasibility of generating test sequences for real-life classification tree models when rapid delivery is not actually a requirement and when more powerful computers are available.
Chapter 7

Conclusions and Future Work

In this chapter we will reflect upon the work that has been done and give an overview of the results of this thesis. Finally, we will discuss interesting open issues and give pointers for possible future research.

7.1 Conclusions

In this thesis project we have investigated the classification tree method and its CTE XL Professional toolset developed by Berner & Mattner Systemtechnik GmbH. We have seen that using this toolset, it is easy to define classification trees and the accompanying test cases and test sequences in a very structured and user-friendly way. Validation and combinatorial generation of test cases is possible by use of dependency and generation rules, however this was not yet possible for validation and generation of test sequences. This is why we stated our main research question: How can we, using user-specified validation and generation rules, perform automatic validation and generation of test sequences for classification trees?

7.1.1 Research questions

The first subquestion related to this research question is: What rules should the end user be able to specify for test sequence validation and generation?

In our research we found that temporal properties that are used extensively in model checking are also a good approach for extending the existing dependency rules in CTE XL. Because the existing Boolean constructs can be retained, it also provides the end user with a relatively small learning curve and good familiarity compared to the classic dependency rules. For generation rules, we provided an exhaustive and deterministic as well as a random and nondeterministic method for generating test sequences, and a set of filter rules that can be specified to limit the amount of generated test sequences. This approach is easy to understand and provides for ample options to influence the results.

Furthermore, we compared different methods for modeling these test sequences using quality criteria such as expressiveness, usability and scalability. We concluded that a decision tree based approach to modeling test sequences is a good compromise between a
7. CONCLUSIONS AND FUTURE WORK

Usable model and adding more expressiveness to test sequences. We then created a tree- and graph-based visualization in CTE XL for these dependency tree based test sequences. For test sequence validation, the labeling method for LTL validation proved to be an accurate method for validation of existing test sequences. However, the rule checking feature of CTE XL re-evaluates all existing test sequences every time a change is made in a classification tree marking, which slowed down the application significantly when a lot of test sequences were in memory.

The second subquestion related to this problem statement is: How can these rules be translated into a complete set of test sequences satisfying these rules?

To answer this question, we implemented algorithms for the generation of test sequences as part of our CTE XL plugin, and performed a series of benchmarks to evaluate the effectiveness and scalability of these rules. We can conclude that these algorithms do not scale very well for problems that involve large amounts of distinct test steps or long test sequences, and that it is hard to achieve a good sequence or transition coverage result in a timely manner. An exception to this is the achievement of transition coverage when doing an exhaustive depth-first generation of test sequences, where a relatively small number of generated test sequences already contain all possible state transitions.

7.2 Contributions

- A survey has been made of models that could be suitable for modeling test sequences for classification trees.
- We have created a new language grammar which adds LTL and CTL operators for validating existing test sequences in CTE XL.
- A CTE XL plugin has been made that can visualize, validate and generate decision tree-based test sequences, and optionally benchmark the generation process.

7.3 Open issues and future work

During the implementation of the validation functionality for test sequences we noticed that the rule checker response becomes very slow when a lot of test sequences are in memory. The reason for this is that with every change of marking in a combination table all test sequences are being re-evaluated instead of just the one that has been changed. Because this concerns the implementation within CTE XL itself it falls outside the scope of this thesis, but this could be improved.

Secondly, more research can be done on more advanced heuristic algorithms for test sequence generation. Especially the generation of test sequences that conform to a certain temporal dependency rule could be very slow because we first generate a sequence and then test whether or not it meets the specification. It is interesting to try and find an advanced temporal algorithm that can invalidate parts of, and therefore many, sequences before the entire sequences have been generated. We expect this to not be easy, because it would be
required to determine which part of the sequence(s) are affected by which part of a temporal logic rule, which we suspect to be a NP-complete problem.

Using these improved algorithms, a more elaborate case study using an existing real-world classification tree model could be executed in order to see to what extent this improves our results. Of particular interest would be the amount of time needed to execute a real-world scenario, and establishing the run times that would still be accepted by the end user of the CTE XL toolset.
Bibliography


Appendix A

Metrics
## A. Metrics

<table>
<thead>
<tr>
<th></th>
<th>status quo</th>
<th>decision trees</th>
<th>timed decision trees</th>
<th>flowcharts Jackson</th>
<th>timed flowcharts Jackson</th>
<th>part nets</th>
<th>timed part nets</th>
<th>PFM</th>
<th>timed PFM</th>
<th>Petri nets</th>
<th>timed Petri nets</th>
<th>Harel FSN</th>
<th>timed Harel FSN</th>
<th>Result (sum)</th>
<th>Result (normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Time</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Loops</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Subprogram</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Synchronization</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Parallels</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td>Determinism</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>2.0</td>
<td>0.29</td>
</tr>
<tr>
<td><strong>Result (sum)</strong></td>
<td>2.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td><strong>Result (normalized)</strong></td>
<td>0.29</td>
<td>0.29</td>
<td>0.43</td>
<td>0.57</td>
<td>0.57</td>
<td>0.71</td>
<td>0.63</td>
<td>0.57</td>
<td>0.60</td>
<td>0.57</td>
<td></td>
<td>0.60</td>
<td></td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td><strong>Aesthetics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Result (sum)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Acquirability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Result (sum)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Result (sum)</strong></td>
<td>2.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td><strong>Result (normalized)</strong></td>
<td>0.29</td>
<td>0.29</td>
<td>0.43</td>
<td>0.57</td>
<td>0.57</td>
<td>0.71</td>
<td>0.63</td>
<td>0.57</td>
<td>0.60</td>
<td>0.57</td>
<td></td>
<td>0.60</td>
<td></td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td><strong>Grand total</strong></td>
<td>7.40</td>
<td>7.55</td>
<td>7.50</td>
<td>6.11</td>
<td>6.05</td>
<td>2.89</td>
<td>2.83</td>
<td>5.80</td>
<td>5.74</td>
<td>3.31</td>
<td></td>
<td>3.10</td>
<td></td>
<td>3.10</td>
<td>3.10</td>
</tr>
</tbody>
</table>

Figure A.1: Metrics
Appendix B

Grammar definitions

B.1 LTL dependency rule grammar

grammar LTL;

options {
  output=AST;
  ASTLabelType=CommonTree;
}

/*
 * This file contains the ANTLR grammar specification of an LTL expression language
 * for restricting certain orders of markings in the combination table.
 * A lexer and parser is generated from this specification.
 *
 * @author Henno Schooljan
 */

/*================================================================--
 * HEADERS
 *================================================================--*/
@header {
package com.berner_mattner.cte.testsequence.decisiontree.model;

import com.berner_mattner.cte.cteObject;
import com.berner_mattner.cte.testsequence.decisiontree.exceptions.NodeNameNotFoundException;
import com.berner_mattner.cte.testsequence.decisiontree.helpers.DecisionTreeHelper;
}

@lexer::header {
package com.berner_mattner.cte.testsequence.decisiontree.model;
}

@members {

B. Grammar definitions

private cteObject cteObj;

public LTLParser(TokenStream input, cteObject cteObj) {
    this(input);
    this.cteObj = cteObj;
}

@Override
public Object recoverFromMismatchedSet(IntStream input, RecognitionException e, BitSet follow)
    throws RecognitionException
{
    throw e;
}

@Override
public Object recoverFromMismatchedToken(IntStream input, int ttype, BitSet follow)
    throws RecognitionException
{
    throw new MismatchedTokenException(ttype, input);
}

@lexer::members

@Override
public void reportError(RecognitionException e) {
    Thrower.sneakyThrow(e);
}

static class Thrower {
    private static Throwable t;
    private Thrower() throws Throwable {
        throw t;
    }

    public static synchronized void sneakyThrow(Throwable t) {
        Thrower.t = t;
        try {
            Thrower.class.newInstance();
        } catch (InstantiationException e) {
            throw new IllegalArgumentException(e);
        } catch (IllegalAccessException e) {
            throw new IllegalArgumentException(e);
        } finally {
            Thrower.t = null; // Avoid memory leak
        }
    }
}

52
B.1. LTL dependency rule grammar

@rulecatch |
catch (RecognitionException e) {
throw e;
}
|
/*------------------------------------------------------------------
* PARSER RULES
*------------------------------------------------------------------*/

rule : stat EOF -> stat;
stat: expr_temp_binary (';'!)?;

expr_temp_binary
: expr_then {'UNTIL'|'RELEASE'} expr_then*;
expr_then: expr_onlythen 'THEN' expr_onlythen*;
expr_onlythen: expr_xor 'ONLYTHEN' expr_xor*;
expr_xor: expr_or 'XOR' expr_or*;
expr_or: expr_and 'OR' expr_and*;
expr_and: expr_temp_unary 'AND' expr_temp_unary*;

expr_temp_unary
: (NEXT|GLOBALLY|FINALLY)? expr_not;
expr_not: (NOT)? atom;

atom: expr_paren
| ID
| if (!DecisionTreeHelper.treeHasName(cteObj, $ID.text)) {
throw new NodeNameNotFoundException(input, $ID.text);
}
|
| STRING
| if (!DecisionTreeHelper.treeHasName(cteObj, $STRING.text)) {
throw new NodeNameNotFoundException(input, $STRING.text, true);
}
|
| INT
| TRUE
| FALSE

53
B. Grammar Definitions

expr_paren
: expr_paren_not // implicitly parenthesized unary expression
| expr_paren_next // implicitly parenthesized unary expression
| expr_paren_always // implicitly parenthesized unary expression
| expr_paren_eventually // implicitly parenthesized unary expression
| '('! expr_temp_binary ')'!; // explicitly parenthesized expression

expr_paren_not: NOTˆ expr_temp_binary;
expr_paren_next: NEXTˆ expr_temp_binary;
expr_paren_always: GLOBALLYˆ expr_temp_binary;
expr_paren_eventually: FINALLYˆ expr_temp_binary;

/*------------------------------------------------------------------
* LEXER RULES
*------------------------------------------------------------------*/

// Boolean constants
TRUE: 'true';
FALSE: 'false';

// Propositional logic symbols
NOT : 'NOT';
AND : 'AND';
OR : 'OR';
XOR : 'XOR';
THEN : '=>';
ONLYTHEN: '<=>';

// Temporal operators
NEXT : 'NEXT';
GLOBALLY: 'GLOBALLY';
FINALLY: 'FINALLY';
UNTIL : 'UNTIL';
RELEASE : 'RELEASE';

// Path quantifiers
ALL : 'ALL';
SOME: 'SOME';

// Other characters
PAREN_LEFT : '(';
PAREN_RIGHT : ')';
STMT_END : ';';
QUOTE : '"';

// Complex tokens
B.2 CTL dependency rule grammar

INT : DIGIT+;
ID : ALPHA (ALPHANUM | '_')*;
STRING: '"' STRINGTEXT '"' {setText($STRINGTEXT.text); };
WHITESPACE : WS_CHAR+ { $channel = HIDDEN; };

// Fragments
fragment DIGIT : '0'..'9';
fragment ALPHA : 'a'..'z' | 'A'..'Z';
fragment ALPHANUM : ALPHA | DIGIT;
fragment STRINGTEXT : (ALPHANUM | '_' | WS_CHAR )+;
fragment WS_CHAR : '	' | ' ' | '' | '
'| '';

grammar CTL;

options {
output=AST;
ASTLabelType=CommonTree;
}

/*
This file contains the ANTLR grammar specification of an CTL expression language
for restricting certain orders of markings in the combination table.
A lexer and parser is generated from this specification.
*
@author Henno Schooljan
*/

/*@header {
package com.berner_mattner.cte.testsequence.decisiontree.model;
import com.berner_mattner.cte.cteObject;
import com.berner_mattner.cte.testsequence.decisiontree.exceptions.NodeNameNotFoundException;
import com.berner_mattner.cte.testsequence.decisiontree.helpers.DecisionTreeHelper;
}
@lexer::header {
package com.berner_mattner.cte.testsequence.decisiontree.model;
}
@members {
private cteObject cteObj;
*/
public CTLParser(TokenStream input, cteObject cteObj) {
    this(input);
    this.cteObj = cteObj;
}

@Override
public Object recoverFromMismatchedSet(IntStream input, RecognitionException e, BitSet follow)
    throws RecognitionException
{
    throw e;
}

@Override
public Object recoverFromMismatchedToken(IntStream input, int ttype, BitSet follow)
    throws RecognitionException
{
    throw new MismatchedTokenException(ttype, input);
}

@lexer::members {

    @Override
    public void reportError(RecognitionException e) {
        Thrower.sneakyThrow(e);
    }

    static class Thrower {
        private static Throwable t;
        private Thrower() throws Throwable {
            throw t;
        }

        public static synchronized void sneakyThrow(Throwable t) {
            Thrower.t = t;
            try {
                Thrower.class.newInstance();
            } catch (InstantiationException e) {
                throw new IllegalArgumentException(e);
            } catch (IllegalAccessException e) {
                throw new IllegalArgumentException(e);
            } finally {
                Thrower.t = null; // Avoid memory leak
            }
        }
    }

    @rulecatch {

    }

}
B.2. CTL dependency rule grammar

catch (RecognitionException e) {
    throw e;
}

/*------------------------------------------------------------------
  * PARSER RULES
  *------------------------------------------------------------------*/
rule : stat EOF -> stat;
stat : expr_temp_binary (';'!)?;
expr_temp_binary
    : expr_then
      | (ALL_UNTIL | ALL_RELEASE | EXISTS_UNTIL | EXISTS_RELEASE) '('! expr_then (','! expr_then)+ ')'!
    ;
expr_then
    : expr_onlythen (THEN` expr_onlythen)*
    ;
expr_onlythen
    : expr_xor (ONLYTHEN` expr_xor)*
    ;
expr_xor
    : expr_or (XOR` expr_or)*
    ;
expr_or
    : expr_and (OR` expr_and)*
    ;
expr_and
    : expr_unary (AND` expr_unary)*
    ;
expr_unary
    : (NOT`)? atom
    ;
atom
    : expr_paren
      | ID
      | if (!DecisionTreeHelper.treeHasName(cteObj, $ID.text)) {
        throw new NodeNameNotFoundException(input, $ID.text);
      }

B. Grammar definitions

```java
if (!DecisionTreeHelper.treeHasName(cteObj, $STRING.text)) {
    throw new NodeNameNotFoundException(input, $STRING.text, true);
}

| STRING |
| INT    |
| TRUE   |
| FALSE  |

expr_paren
: expr_paren_not // implicitly parenthesized unary expression
| expr_paren_a_next // implicitly parenthesized unary expression
| expr_paren_a_always // implicitly parenthesized unary expression
| expr_paren_a_eventually // implicitly parenthesized unary expression
| expr_paren_e_next // implicitly parenthesized unary expression
| expr_paren_e_always // implicitly parenthesized unary expression
| expr_paren_e_eventually // implicitly parenthesized unary expression
| '(! expr_temp_binary ')'! // explicitly parenthesized expression

expr_paren_not
: NOT^ expr_temp_binary

expr_paren_a_next
: ALL_NEXT^ expr_temp_binary

expr_paren_a_always
: ALL_GLOBALLY^ expr_temp_binary

expr_paren_a_eventually
: ALL_FINALLY^ expr_temp_binary

expr_paren_e_next
: EXISTS_NEXT^ expr_temp_binary

expr_paren_e_always
: EXISTS_GLOBALLY^ expr_temp_binary

expr_paren_e_eventually
: EXISTS_FINALLY^ expr_temp_binary
```
B.2. CTL dependency rule grammar

 /*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

// Boolean constants
TRUE: 'true';
FALSE: 'false';

// Propositional logic symbols
NOT : 'NOT';
AND : 'AND';
OR : 'OR';
XOR : 'XOR';
THEN : '=>';
ONLYTHEN: '<=>';

// Combined Temporal operators
ALL_NEXT : 'AX';
ALL_GLOBALLY: 'AG';
ALL_FINALLY: 'AF';
ALL_UNTIL: 'AU';
ALL_RELEASE: 'AR';

EXISTS_NEXT : 'EX';
EXISTS_GLOBALLY: 'EG';
EXISTS_FINALLY: 'EF';
EXISTS_UNTIL: 'EU';
EXISTS_RELEASE: 'ER';

// Other characters:
PAREN_LEFT: '(';
PAREN_RIGHT: ')';
STMT_END: ';';
QUOTE: '"';
COMMA: ',';

// Complex tokens
INT : DIGIT+;
ID : ALPHA (ALPHANUM | '_')*;
STRING: '"' STRINGTEXT '"' {setText($STRINGTEXT.text); };
WHITESPACE : WS_CHAR+ { $channel = HIDDEN; };

// Fragments
fragment DIGIT : '0'..'9';
fragment ALPHA : 'a'..'z' | 'A'..'Z';
B. Grammar definitions

fragment ALPHANUM : ALPHA | DIGIT;
fragment STRINGTEXT : (ALPHANUM | '_' | WS_CHAR )+;
fragment WS_CHAR : '\t' | ' ' | '\r' | '\n' | '\000c';
Appendix C

ATM state model

Figure C.1: ATM state model
## Appendix D

### Benchmark Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MaxTime</th>
<th>GroupSize</th>
<th>MinSteps</th>
<th>MaxSteps</th>
<th>Duration</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>0.02</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>0.05</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>6</td>
<td>0.15</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>6</td>
<td>0.39</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>16</td>
<td>1</td>
<td>6</td>
<td>0.89</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>6</td>
<td>1.85</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>6</td>
<td>3.55</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>22</td>
<td>1</td>
<td>6</td>
<td>6.3</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>10.69</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table D.1: Brute force, full coverage, durations with group sizes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MaxTime</th>
<th>GroupSize</th>
<th>MinSteps</th>
<th>MaxSteps</th>
<th>Duration</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>0.05</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0.43</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>4.07</td>
<td>-1</td>
</tr>
<tr>
<td>Brute force</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>34.34</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table D.2: Brute force, full coverage, durations with sequence lengths

63
## D. Benchmark Results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>maxTime</th>
<th>groupSize</th>
<th>minSteps</th>
<th>maxSteps</th>
<th>duration</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>0.02</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>0.06</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>12</td>
<td>1</td>
<td>6</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>14</td>
<td>1</td>
<td>6</td>
<td>0.53</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>16</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>83.59</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>18</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>40</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>20</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>20.94</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>22</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>11.46</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>24</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>6.9</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>26</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>4.19</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>28</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>2.69</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>30</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>1.76</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>32</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>1.18</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>34</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.81</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>36</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.57</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>38</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.52</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>40</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.52</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>42</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.51</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>44</td>
<td>1</td>
<td>6</td>
<td>1.02</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table D.3: Brute force, sequence coverage on 1 second with group sizes

<table>
<thead>
<tr>
<th>algorithm</th>
<th>maxTime</th>
<th>groupSize</th>
<th>minSteps</th>
<th>maxSteps</th>
<th>duration</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>0.01</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>0.06</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>0.58</td>
<td>100</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>1.02</td>
<td>19.43</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>1.02</td>
<td>2.17</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1.02</td>
<td>0.49</td>
</tr>
<tr>
<td>Brute force</td>
<td>1000</td>
<td>10</td>
<td>1</td>
<td>11</td>
<td>1.02</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Table D.4: Brute force, sequence coverage on 1 second with sequence lengths
<table>
<thead>
<tr>
<th>algorithm</th>
<th>maxTime</th>
<th>groupSize</th>
<th>minSteps</th>
<th>maxSteps</th>
<th>duration</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>2000</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>72.13</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>6</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>1.82</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.09</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>12</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>14</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>16</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>18</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>20</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table D.5: Random, sequence coverage on 2 seconds with group sizes

<table>
<thead>
<tr>
<th>algorithm</th>
<th>maxTime</th>
<th>groupSize</th>
<th>minSteps</th>
<th>maxSteps</th>
<th>duration</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>2.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>2.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>2.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>2.02</td>
<td>98.14</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>6</td>
<td>2.02</td>
<td>32.73</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>2.02</td>
<td>5.3</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>2.02</td>
<td>0.69</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>2.02</td>
<td>0.08</td>
</tr>
<tr>
<td>Random</td>
<td>2000</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>2.02</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table D.6: Random, sequence coverage on 2 seconds with sequence lengths
### D. Benchmark Results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>maxTime</th>
<th>groupSize</th>
<th>minSteps</th>
<th>maxSteps</th>
<th>duration</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>500</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>14</td>
<td>1</td>
<td>5</td>
<td>0.01</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>16</td>
<td>1</td>
<td>5</td>
<td>0.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>18</td>
<td>1</td>
<td>5</td>
<td>0.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>20</td>
<td>1</td>
<td>5</td>
<td>0.02</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>22</td>
<td>1</td>
<td>5</td>
<td>0.03</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>24</td>
<td>1</td>
<td>5</td>
<td>0.05</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>26</td>
<td>1</td>
<td>5</td>
<td>0.05</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>28</td>
<td>1</td>
<td>5</td>
<td>0.08</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>30</td>
<td>1</td>
<td>5</td>
<td>0.1</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>32</td>
<td>1</td>
<td>5</td>
<td>0.12</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>34</td>
<td>1</td>
<td>5</td>
<td>0.16</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>36</td>
<td>1</td>
<td>5</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>38</td>
<td>1</td>
<td>5</td>
<td>0.25</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>40</td>
<td>1</td>
<td>5</td>
<td>0.32</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>42</td>
<td>1</td>
<td>5</td>
<td>0.36</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>44</td>
<td>1</td>
<td>5</td>
<td>0.45</td>
<td>100</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>46</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>98.58</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>48</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>85.77</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>50</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>75.95</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>52</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>67.86</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>54</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>58.34</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>56</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>52.58</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>58</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>46.63</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>60</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>41.4</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>62</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>37.85</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>64</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>33.28</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>66</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>29.66</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>68</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>26.92</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>70</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>24.11</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>72</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>22.06</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>74</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>19.57</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>76</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>17.86</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>78</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>16.39</td>
</tr>
<tr>
<td>Random</td>
<td>500</td>
<td>80</td>
<td>1</td>
<td>5</td>
<td>0.52</td>
<td>15.13</td>
</tr>
</tbody>
</table>

Table D.7: Random, transition coverage on 0.5 seconds with group sizes
Table D.8: Random, transition coverage on 0.5 seconds with sequence lengths