Enhanced Test Case Generation with the Classification Tree Method

Inauguraldissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von
Peter Michael Kruse

Berlin 2013
D 188

Betreuerin:	Prof. Dr.-Ing. Ina Schieferdecker
Erstgutachterin:	Prof. Dr.-Ing. Ina Schieferdecker
Zweitgutachter:	Prof. Dr. rer. nat. habil. Bernd-Holger Schlingloff
Drittgutachter:	Prof. Dr. phil.-nat. Jens Grabowski

Tag der Disputation: 25. August 2014

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ina Schieferdecker and Joachim Wegener, for their professional guidance, inspiring discussions, and encouragement during the period of this research. Furthermore, I would like to thank all my colleagues from Berner & Mattner Systemtechnik GmbH. Thanks also go to my friends, my parents, and my family for their support and encouragement. Special thanks go to Stefanie Winzek and Malte Zander for numerous helpful comments, hints, and suggestions on early versions of this work.

Abstract

To make a statement on software quality, a methodical approach on software testing is absolutely necessary. One common approach is the classification tree method introduced in 1993. All relevant test aspects of a system under test and their characteristics are divided into disjoint subsets. Test cases are then generated by combining specific characteristics of each aspect. Depending on the size (cardinality) and number of different aspects, the number of possible combinations grows exponentially. Therefore tools for applying the classification tree method offer coverage criteria for automated test case generation. Typically current coverage criteria are minimal or complete combination. Additionally, test case generation needs to consider specific dependencies between characteristics of test aspects.

Existing approaches do not offer a prioritization of certain test aspects during test case generation. However, prioritization of test aspects is important for a number of different reasons: Test resources may be limited, testing can be expensive, as it may be destructive, or it may be desirable to only use a subset of test cases as an initial test.

Current approaches can be divided into deterministic and non-deterministic techniques. The current classification tree editor uses a non-deterministic technique to generate test cases. Resulting test suites vary in size and composition. During test case generation, dependencies and the classification tree itself are stored in different data structures. Therefore, test cases additionally need to be validated against the dependency rules during test case generation.

Up to now, no approach supports the automated generation of test sequences from classification trees. Test sequences can however be defined manually by the user.

This work presents a new approach for qualifying the classification tree with test aspects of importance (e.g. test costs, test duration, probabilites etc.). These numbers can be used for prioritized test case generation and to optimize test suites and, therefore, to reduce their size.

For dependency handling, we use an integrated data structure holding both the classification tree and its dependency rules. The data structure is also used for a new deterministic test case generation, which handles dependencies directly during the process of test case generation. The resulting test suite should be equal or smaller in size while generation should be as fast as or even faster than current generation approaches.

Finally, a new approach for automatic test sequence generation from classification trees is presented as well. We identify parameters for test sequence generation and develop new dependency rules and new generation rules.

Results are then compared using common algorithms and standard benchmarks.

Zusammenfassung

Um eine Aussage über die Qualität von Software machen zu können, sind methodische Ansätze für den Softwaretest dringend erforderlich. Ein typischer Ansatz ist die 1993 vorgestellte Klassifikationsbaum-Methode. Bei ihr werden alle testrelevanten Aspekte eines Testsystems und seine Eigenschaften in disjunkte Teilmengen zerlegt. Testfälle werden dann aus der Kombination von spezifischen Charakteristika aller Testaspekte gebildet. Je nach Anzahl der Testaspekte und der Menge der enthaltenen Elemente wächst die Anzahl der möglichen Kombination exponentiell. Werkzeuge zur Unterstützung der Klassifikationsbaum-Methode bieten daher Abdeckungskriterien für die automatische Testfallgenerierung an. Typische Abdeckungskriterien sind die Minimalkombination oder die vollständige Kombination. Darüber hinaus werden Abhängigkeiten zwischen Testaspekten berücksichtigt. Priorisierung von Testaspekten während der Testfallgenerierung bietet bislang keiner der bestehenden Ansätze, dabei wäre diese aus einer Reihe von Gründen sehr wichtig: Testressourcen sind in der Regel begrenzt und Testen ist kostenintensiv, insbesondere beim destruktiven Testen. Auch kann es gewünscht sein, nur eine Untermenge aller Testfälle als Eingangstest zu nutzen.

Bestehende Ansätze zur Testfallgenerierung lassen sich in zwei Gruppen unterteilen, deterministische und nicht-deterministische Techniken. Der aktuelle Klassifikationsbaum-Editor generiert Testfälle nicht-deterministisch, so dass resultierende Testsuiten aus verschiedenen Durchläufen in Größe und Zusammenstellung variieren. Außerdem werden Klassifikationsbaum und Abhängigkeitsregeln in unterschiedlichen Datenstrukturen gespeichert, so dass Testfälle umständlich auf Gültigkeit gegen die Abhängigkeitsregeln geprüft werden müssen.

Keiner der bestehenden Ansätze unterstützt die automatische Generierung von Testsequenzen aus Klassifikationsbäumen. Testsequenzen können bislang nur manuell definiert werden.

In dieser Arbeit präsentieren wir einen neuen Ansatz um Klassifikationsbäume mit Gewichten (Testkosten, Testdauer, Wahrscheinlichkeiten, usw.) zu qualifizieren. Die Gewichte werden von der priorisierenden Testfallgenerierung genutzt. So kann die resultierende Testsuite optimiert und ihr Umfang reduziert werden.

Abhängigkeitsregeln werden zusammen mit dem Klassifikationsbaum in einer Datenstruktur gespeichert. Diese wird auch für die deterministische Testfallgenerierung genutzt, welche die Abhängigkeitsregeln direkt während der Erzeugung von Testfällen berücksichtigt. Im Vergleich zur bisherigen Testfallgenerierung darf die hierbei entstehende Testsuite weder größer sein noch darf ihre Erzeugung länger dauern.

Schließlich präsentieren wir einen neuen Ansatz zur automatischen Generierung von Testsequenzen aus Klassifikationsbäumen. Wir identifizieren Parameter für die Generierung und neue Abhängigkeitsregeln.

Alle Ergebnisse werden in standardisierten Benchmarks mit anderen Algorithmen verglichen.

Declaration

The work presented in this thesis is original work undertaken between January 2009 and November 2013 at Berner & Mattner Systemtechnik GmbH. Portions of this work have been published elsewhere:

- Peter M. Kruse and Magdalena Luniak: *Automated Test Case Generation Using Classification Trees* in Proceedings of StarEast 2010, Orlando, Florida, USA, 2010. Best Paper Award. Concerning Prioritized Test Case Generation [KL10a].
- Peter M. Kruse and Magdalena Luniak: *Automated Test Case Generation Using Classification Trees* in Software Quality Professional, Volume 13, Issue 1, issued by the American Society for Quality (ASQ), 2010. Concerning Prioritized Test Case Generation [KL10b].
- Peter M. Kruse and Joachim Wegener: *Sequenzgenerierung aus Klassifikationsbäumen* in Softwaretechnik-Trends, Band 31, Ausgabe 1, as part of the Proceedings zum 31. Treffen der Fachgruppe TAV der Gesellschaft für Informatik, Paderborn, Germany, 2011. Concerning Test Sequence Generation [KW11a].
- Peter M. Kruse and Joachim Wegener: *Test Sequence Generation from Classification Trees* in Sistemas e Tecnologias de Informação, Actas da 6ª Conferência Ibérica de Sistemas e Tecnologias de Informação (CISTI 2011), Chaves, Portugal, 2011 [KW11b].
- Peter M. Kruse and Kiran Lakhotia: *Multi Objective Algorithms for Automated Generation of Combinatorial Test Cases with the Classification Tree Method* in Symposium on Search Based Software Engineering (SSBSE 2011), Szeged, Hungary, 2011 [KL11].
- Peter M. Kruse: *Test Sequence Generation from Classification Trees using Multi-agent Systems* in Proceedings of 9th European Workshop on Multi-agent Systems (EUMAS 2011), Maastricht, the Netherlands, 2011 [Kru11].
- Peter M. Kruse and Joachim Wegener: *Test Sequence Generation from Classification Trees* in Proceedings of ICST 2012 Workshops (ICSTW 2012), Montreal, Canada, 2012 [KW12].
- Javier Ferrer and Peter M. Kruse and J. Francisco Chicano and Enrique Alba: *Evolutionary Algorithm for Prioritized Pairwise Test Data Generation* in Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2012, Philadelphia, USA, 2012 [FKCA12].
- Peter M. Kruse and Ina Schieferdecker: *Comparison of Approaches to Prioritized Test Generation for Combinatorial Interaction Testing*, in Proceedings of Federated Conference on Computer Science and Information Systems (FedCSIS) 2012, Wroclaw, Poland, 2012 [KS12].

Portions of this work have been published as part of EU-ICT STREP FP7-ICT-2009-5 257574 FITTEST (Future Internet Testing) deliverables:

- Peter M. Kruse: *D5.1 - Report on the generation of Classification Trees*, March 2012.
- Peter M. Kruse: *D5.2 - Interface Implementation to Internet Testing Infrastructure and Extended User Interface*, October 2012.
- Peter M. Kruse: *D5.3 - Report on Test Suite Optimization and New Test Case Generation Techniques*, May 2013.

This work is supported by diploma and master theses instigated and supervised by me:

- Magdalena Luniak: *Priorisierende Kombinationsregeln in der Klassifikationsbaum-Methode* (Prioritized Combination Rules for the Classification Tree Method), Diploma thesis, TU Berlin, 2009 [Lun09].
- Robert Reicherdt: *Testfallgenerierung mit Binary Decision Diagrams für Klassifikationsbäume mit Abhängigkeiten* (Test Case Generation with Binary Decision Diagrams for Classification Trees with Dependency Rules), Master thesis, TU Berlin, 2010 [Rei10].
- Nick Walther: *Testsequenz-Generierung und Repräsentation mit der Klassifikationsbaum-Methode* (Test Sequence Generation and Representation with the Classification Tree Method), Diploma thesis, HU Berlin, 2011 [Wal11].
- Henno Schooljan: *Test Sequence Validation and Generation using Classification Trees*, Master thesis, TU Delft, 2013 [Sch13].

Curriculum Vitae

The CV is not included in the online version for reasons of privacy.

Der Lebenslauf ist in der Online-Version aus Gründen des Datenschutzes nicht enthalten.

Contents

1 Introduction

Software has become a central part of our everyday life, both visible to and hidden from human notice. Software controls alarm clocks, coffee and washing machines, the garage door, or the house alarm system. Car engines have a software-driven engine control unit. Car radios use software to play music files from a wirelessly connected mobile phone, which itself contains software. Traffic lights and intelligent traffic signs are controlled by software, as well as GPS devices used to guide our ways. There is already software inside the human body as part of pacemakers or intelligent prostheses.

While there are many benefits from the presence of software in every day life, there are also risks. Software malfunction can lead to discomfort and loss of money. In some applications, malfunction of software may even be life-threatening.

Therefore, it is essential for the producers and developers of software to avoid all kinds of malfunction whenever possible. Avoiding errors in software development is usually performed using quality assurance. Quality assurance in software development normally contains a defined software development processes, a major aspect of which is software testing.

Software testing is used to gain levels of confidence in the software quality. So while testing generally speaking cannot prove the absence of errors, it can show that the software is performing in accordance with its specification for tested scenarios. The software specification usually consists of functional and non-functional requirements. While software testing can be set up and performed easily, it is still time-consuming and labor intensive.

Other approaches for ensuring software quality include formal reviews and mathematical proofs, but both of them tend to be even more laborious and complex in application.

Obviously, it is desirable to reduce testing efforts to a feasible minimum to get good test results with reasonable efforts and within resource constraints. Therefore, there are several approaches for test case design and test case selection.

One common approach is the equivalence class partitioning that derives valid and invalid partitions for input data from the functional specification. For each partition, only one representative is tested as all values from one partition range are considered to result in the same outcome. From the broad range of all possible values, the determination of only one representative per partition can lead to a tremendous reduction of test effort.

The combinatorial interaction test design is based on the observation that many errors occur due to interactions between parameters. Therefore, it is crucial to select

test cases combining as many different parameters as possible and to avoid redundant test cases, which recombine parameters already combined in earlier test cases.

The classification tree method is a common approach based on the equivalence partitioning and can be used for both test planning and test design. It allows for a systematic specification of the system under test and its corresponding test cases. The classification tree editor implements the combinatorial test design and allows to automatically generate test suites for given coverage levels.

1.1 Motivation

The classification tree method is supported by a graphical editor, the classification tree editor. The editor adopts results from the field of combinatorial interaction testing, which allows generating certain test suites automatically once the system under test has been specified.

Some test cases can be more important than others. To identify the importance of a test case, it is necessary to make a statement on the order of test cases. For different test aspects, there might be different values of importance. For the test aspect *costs*, these values might be Euro values, for the test aspect *duration*, it would be some time unit. Having an order of importance it would be possible to select only the most important test cases in accordance with standards or available resources. So for software with low standards, e.g. a multimedia player, a smaller set of test cases might be sufficient in contrast with a pacemaker, where larger sets of test cases would be desirable. For time constraints, the same considerations are applicable, e.g. if less time is available for regression testing, then only a subset of the most important n test cases could be performed while for initial and final testing, a larger set of test cases may be considered. Other aspects of importance would be occurrence and failure probabilities. Having a test suite containing test cases in order of occurrence probability, testers could use the most probable configuration as an initial test and continue testing only if this first test cases passes. Otherwise, it could be interesting to start with the least probable configuration and test how thorough all infrequent scenarios have been considered. The same considerations apply for failure probabilities as well.

Currently, the classification tree method does not provide measures of importance nor does the classification tree editor use these measures to sort test cases.

For test case selection, the use of coverage criteria is quite common. The classification editor can create test suites for given coverage levels, e.g. pairwise coverage. There is also support for constraints. As there might be combinations of parameters which cannot occur at the same time, it is possible to exclude them from test case generation and to check manually-created test cases against dependency rules. One advantage of the classification tree method is its replicability. Classification trees can easily be reviewed. All test aspects are represented as classifications in the classification tree; all possible instances for these aspects are classes under corresponding classifications. The replicability of test cases is, however, limited. The classification

tree editor allows for specifying generation rules for the test case generator which then generates test cases based on a random process. So while each run generates a test suite that fulfills the generation rule with desired coverage levels, results from different generation runs may vary. It is therefore not possible to repeat a certain test case generation and obtain exactly the same results. Certain industry standards (e.g. ISO 26262), however, require tractability of test cases and their design. So while a test suite generated with the current classification tree editor fulfills coverage levels, later manipulations to the test suite cannot be revealed. Additionally, the current classification tree editor performs badly with complex dependency rules or compositions of contradictory rules.

It would therefore be desirable to have a deterministic test case generation which produces exactly the same result for the same generation problems and handles dependency rules neatly.

Usually, software is used continuously. After performing one task, there are typically further things to do: A media player plays one track after another, car navigation leads to the destination and gives directions for several intersections and junctions, and washing machines perform several actions as part of the washing cycle. For testing these different continuous actions, test cases must therefore reflect these elements in a certain order. The outcome of one test case is used as the input for the next test case. The composition of several test cases into a larger test scenario is called test sequence. The elements of the test sequence are called test steps. Test sequences can then be used to model consecutive events. For practical reasons, it is rarely possible to test all possible test sequences, so test sequence design is a crucial task. In addition to countless configurations for single test steps, there are potentially endless different orders and repetitions of test steps. Furthermore, test steps cannot be composed in any arbitrary order as it is required for some configurations of the software that other things have been done first.

Currently, it is possible to manually specify test sequences in the classification tree editor. There is, however, no predefined way of specifying constraints between test steps of a test sequence. There are no measures of coverage and there is no automatic generation of test sequences available with the classification tree method.

So while the classification tree method and the corresponding editor are of great help for test engineers, there are still a number of shortcomings:

1. The classification tree method and its editor do not allow the prioritization of certain test aspects and do not offer prioritized test case generation either.

2. The current classification tree editor offers only limited functionality for automated test case generation. Complex systems with constraints, for example, are processed only slowly and test results cannot be reproduced, due to the random generation process. Moreover, automated test case generation is prone to errors.

3. Test sequences can be defined only manually. There is neither a concept for

dependency rules between single test steps nor automated test sequence generation in the classification tree editor.

1.2 Goal

The goal of this work is to handle each issue identified in the previous section and develop a solution or at least an improvement for it.

To allow the prioritization of test aspects and the prioritized test case generation, we develop prioritization models to be integrated into the classification tree. This allows the tester to assign priority values to elements of the classification tree. We then extend existing combination rules for test case generation using these priority values. We also develop new combination rules taking these values into account for statistical testing. Resulting test cases then have a defined importance for the test suite resulting from the priorities assigned to elements of the classification tree used for this test case. Additionally, the introduction of coverage measures allows measuring coverage level of generated test cases. All generation rules still support dependency rules specified for the system under test in the classification tree. This allow testers to generate test suites with an order of importance specified by the priority values assigned to classification tree elements and to select subsets of test suites that have a defined quality, calculated using coverage rates with the new coverage measures.

The handling of dependency rules during test case generation is not ideal in existing approaches for test case generation with the classification tree method. Also, for reasons of reproducibility, it can be desirable to use deterministic test case generation. Therefore, the goal is to develop a new approach for dependency rule handling and a new deterministic approach for test case generation with constraints using the innovative dependency handling. The goal is to develop a unified representation for both the dependency rules and the classification tree, in order to ease the handling of constraints. This unified representation is then used to develop deterministic test case generation, which must support generation rules to specify coverage levels for resulting test suites and should be as efficient as or even more efficient than current approaches. The latter requires the ability to generate test cases for a subset of the classification tree as well as the support for the current granularity of combination rules. Implicit dependencies, given by refinements in the classification tree, must be respected. Test case generation must provide only valid test cases, of course. If there already are test cases (e.g. from manual definition), it should be possible to generate a test suite containing these test cases and still complete the coverage level defined by the generation rule.

For testing the continuous behavior of the system with the classification tree method, we must develop a way to allow for the generation of test sequences. Advanced dependency rules are introduced to model continuous behavior of the system under test. The advanced dependency rules allow the tester to specify the legal transitions of different states of the system. Then the test sequence generation produces test suites of test sequences that fulfill the advanced generation rules specifying desired cover-

age level. New coverage levels for continuous systems are to be defined. Of course, the test steps of generated test sequences still have to comply with the conventional dependency rules already known for test case generation.

1.3 Approach

For the prioritized test case generation, we extend the classification tree elements with explicit values of importance, so-called weights. We identify qualification models to provide these weights with a semantic meaning. New combination rules are then used together with the qualified tree for prioritized test case generation, which results in an ordered test suite. The test amount can be optimized with respect to certain quality goals.

The handling of dependency rules is performed using an integrated data structure containing both the classification tree and the dependency rules. Since the data structure represents the classification tree with all integrated dependency rules, it contains only valid test cases.

This integrated data structure is exploited for new, unprioritized test case generation. The tester can specify coverage levels using generation rules.

We identify Binary Decision Diagrams (BDD) as a proper data structure for the integrated representation of dependency rules and the classification tree.

For the test sequence generation, we define new dependency rules which describe constraints between single test steps. New generation rules allow specifying coverage and granularity of the resulting suite of test sequences.

We split our approach into the following steps:

1. Definition of prioritization models, selection of aspects that are important for test set optimization.

2. Qualification of the classification tree by using prioritization models.

3. Development of an algorithm for mapping the set of valid test cases given by a classification tree and its dependencies onto a logical expression.

4. Development of prioritized test case generation rules.

5. Development of a deterministic algorithm for generating test suites matching given coverage criteria expressions.

6. Development of test sequence dependency rules and generation rules.

7. Development of an algorithm for test sequence generation.

8. Evaluation of our approaches using standard benchmarks.

1.4 Results

All goals have been successfully completed. For the prioritized test case generation, we have developed and implemented the prioritized test case generation using qualified classification trees. For qualification, we introduce three difference usage models allowing us to specify the value of importance directly in the classification tree. These weights can be assigned to classification tree elements and are then considered during prioritized test case generation. To handle dependencies in prioritized test case generation, we establish a mapping of the classification trees and dependencies onto a logical expression, representing the set of valid test cases in a Binary Decision Diagram. The prioritized test case generation is then compared against existing approaches. In most cases, our prioritized test case generation performs better than or as well as existing approaches.

A new deterministic test case generation using the integrated representation of classification tree and dependency rules as BDD has been developed and implemented. Handling of dependency rules has been improved over existing approaches for test case generation in the classification method. We have then compared our new approach using a large set of benchmarks and compare our result with both previous test case generation of the classification tree method as well as existing approaches for combinatorial testing. Our approach performs better than previous test case generation with the classification tree method in terms of both result set size and generation time. The performance in comparison with other combinatorial test case generation techniques, though, is not always better in terms of result set size and generation time. For specific tasks, e.g. the smallest test suite generated using a deterministic approach, our new test case generation is still very good.

For the new test sequence generation with the classification tree method we have developed new advanced dependency rules and new advanced test generation rules. The advanced dependency rules allow testers to specify constraints between different steps of a test sequence in addition to existing dependency rules describing constraints between different classifications and classes during a single test step. For the advanced generation rules, we have developed new coverage levels specific to the continuous nature of systems under test. After all approaches and rules had been developed, we have implemented several prototypes using different representation techniques. We needed three attempts until we found a good representation and interpretation of classification trees. Preliminary results from the first two attempts are, however, included to allow a comparison and provide some of the lessons learned. Our third approach for test sequence generation interprets classification trees as statecharts with some simplifications. We then travel the statechart using a multi-agent system. To the best of our knowledge, there are no existing benchmark suites available for continuous testing in combinatorial testing so we had to select a suite of benchmarks on our own. In the identified set of benchmark scenarios, test sequence generation performs well in all scenarios tested.

1.5 Structure

In Chapter 2, we provide the background for this work. We introduce combinatorial testing, the classification tree method and the classification tree editor. Existing test case generation approaches for the classification tree method are presented as well.

Important terms and basics for understanding this work as well as related work are presented in Chapter 3. Several approaches and techniques for combinatorial testing, both greedy and search-based, are described. Then, fundamental work on test sequence generation and validation is presented.

In Chapter 4 we design our enhancements for test case and test sequence generation with the classification tree method. We select three prioritization models and give a prioritization example to then qualify the classification tree. For dependency handling, we transform both the classification tree and its dependency rules into an integrated data structure, a Binary Decision Diagram. Afterwards we use this data structure for dependency handling in prioritized test case generation. Additionally, we exploit the data structure for a new deterministic test case generation as well. We introduce our implementation for test case generation. For our new test sequence generation we design the requirements for both new dependency rules and new generation rules. Actual generation is then done by converting the classification tree into a statechart and traversing it using a multi-agent system.

We evaluate our approaches in Chapter 5. We implement our algorithms in Java 6 and use the classification tree editor tool for prototype integration. This offers direct access to the data model of the classification tree editor, containing the classification tree, generation rules and dependency rules. First, we apply a prioritized benchmark to the prioritized pairwise combination. We then optimize our new deterministic test case generation approach using a small set of standard benchmarks and compare it to the previous test case generation of classification tree editor, too. We then apply a large set of standard benchmarks to the optimized deterministic test case generation approach. Finally, we introduce a set of case studies for test sequence generation and use them to evaluate our new test sequence generation approach.

In Chapter 6, we assess all three approaches. We show advantages and disadvantages and draw conclusions. At the end, we give an overview of future work.

2 Background

In this chapter, we present combinatorial testing, test case generation with coverage criteria and the classification tree method together with the CTE tool. We used CTE to evaluate our results.

2.1 Combinatorial Testing

Combinatorial testing, also called combinatorial interaction testing (CIT [CSR06]) or combinatorial test design (CTD), is a technique that designs tests for a system under test by combining input parameters. For each parameter of the system, a value is chosen. This collection of parameter values is called test case. The set of all test cases constitutes the test suite for the system under test.

For a complete test of any given system, it would be necessary to select all possible values for each parameter and completely combine it with all possible values of the remaining parameters. Testing all possible test cases may prove that the system is fault-free and therefore helps to gain high confidence in system quality. Since the number of resulting test cases grows exponentially with the number of parameters and their possible parameter values, complete combination is only used for very small systems with only few input parameters and possible values or for safety critical systems, such as aircraft control software.

To reduce the amount of test cases without losing too much confidence in the system, several approaches have been developed. These approaches can be divided into two groups: Approaches reducing the number of parameter values and approaches reducing the number of parameter combinations. Approaches from both groups can be combined to further reduce the size of test suites.

Boundary value analysis or *equivalence class grouping*, for example, aim to introduce sections for input parameter values [Mye79]. It is assumed, that each candidate of a section results in the same system behavior. This grouping into classes reduces the number of parameter values and therefore leads to a reduced test suite size.

For the reduction of combinatorial complexity, some approaches introduce coverage criteria. The usage of t-wise coverage is based on the fact that many faults in systems are triggered by the interaction of two or more parameters. For certain errors to show, it is therefore not always necessary to test all possible combinations of parameter values. t-wise coverage can be defined as follows: A test suite fulfills a given t-wise coverage, if it contains each t-wise combination of parameter values at least once [WP01].

2.1.1 Coverage Criteria

We distinguish between three kinds of coverage levels: Minimal coverage, maximal coverage and levels in between.

For **minimal coverage** (with $t = 1$), all values from each parameter need to be included in the resulting test suite at least once. The parameter with the highest number of values determines the resulting test suite size. One test case is needed for each value of this parameter, combined with the values of all the other parameters with a smaller number of values.

For **maximal coverage** (with $t = n$), all values from each parameter are completely combined with values from all the other parameters. The number of test cases in the resulting test suite size can be calculated by forming the Cartesian product of all parameters.

The **remaining coverage levels** are coverage levels with t-wise parameter interaction ($1 < t < n$). These coverage levels are used, because they offer a good compromise in terms of both, test suite size and parameter interaction. The resulting test suite is smaller than a test suite for maximal coverage, while the parameter interaction is better than plain minimal coverage. Their optimal sizes, however, cannot easily be predicted and the calculation of minimal t-wise test suite is NP-complete. Lei and Tei showed this for $t = 2$ [LT98] and Wiliam and Probert proved this for all $t \geq 2$, too [WP01].

Finding a non-minimal test suite for a t-wise coverage level, however, is a trivial task, since the complete combination includes coverage for all levels of $t < n$, too.

2.1.2 Constraints

In real world scenarios some of the possible parameter value combinations may not be valid. Therefore, constraints have been introduced. They allow specifying invalid combinations which will be excluded or skipped during test case generation.

Constraints can be found in many specifications of a software system. They are typically given in natural language and exist for several reasons, such as limitations of the components used in the target system, available resources and even marketing decisions. While constraints reduce the number of valid test cases, their presence makes interaction testing more challenging [CDS07]. The impact of constraints varies with the (test) problem, but their presence causes problems for many existing CIT tools. Of the numerous existing tools supporting combinatorial interaction test design only a few offer full constraints support. Of these few tools those with full published details are even rarer.

Constraints that are expressed explicitly in the system description can give rise to implicit relationships between other choices. In fact, treatment of these implicit relationships is the key complicating factor in solving constrained CIT problems.

In the presence of constraints:

- the number of required t-sets to produce a solution can not be calculated and

- lower and upper bounds of the solution size cannot be calculated.

So while explicitly forbidden tuples can be immediately removed from the t-set of tuples to cover, implicitly forbidden tuples may arise during calculation and evaluation. And while constraints always decrease the number of feasible system configurations, they might both increase and decrease the sample size [CDS07].

2.2 Classification Tree Method

The classification tree method [GG93] (CTM) has been introduced in 1993. It describes a systematic approach to test case design and was inspired by the category partition method [OB88] by Ostrand and Balcer. In the category partition method (CPM) input data and environment parameters are analyzed for characteristics and their influence on the test object.

Applying the classification tree method involves two steps—designing the classification tree and defining test cases.

Designing the classification tree. In the first phase, all aspects of interests and their disjoint values are identified. Aspects of interests, also known as parameters, are called *classifications*, their corresponding parameter values are called *classes*.

Any system under test can be described by a set of classifications, holding both input and output parameters. Each classification can have any number of disjoint classes, describing the occurrence of the parameter. All classifications together form the *classification tree*. For semantic purpose, classifications can be grouped into *compositions*.

Additionally, all classes can be refined using new classifications with classes again.

Figure 2.1 shows an example tree: The test object is a function performing operations on a list. Two aspects of interest have been identified, *sorting* and *list length*, so there are two classifications in the tree.

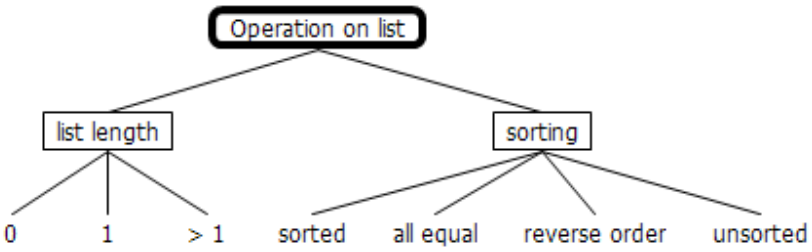

Figure 2.1: Classification Tree for “List Operation” Example

These characteristics will influence the test object behavior. For each classification, the input domain is divided into disjoint subsets, the classes. The significant occurrences *empty*, *one element*, *more than one element* can be identified for *list length* property and *sorted*, *all equal*, *reverse order*, and *unsorted* for the *sorting* property.

Thus, all parameters are classified and categorized.

Definition of test cases. Having composed the classification tree, test cases are defined by combining classes of different classifications. For each classification, a significant representative (class) is selected. Classifications and classes of classifications are disjoint. Since classifications contain only disjoint values—obviously lists cannot be sorted and unsorted at the same time—test cases cannot contain several values of one classification. This small example would result in 12 different combinations to test.

2.3 Classification Tree Editor

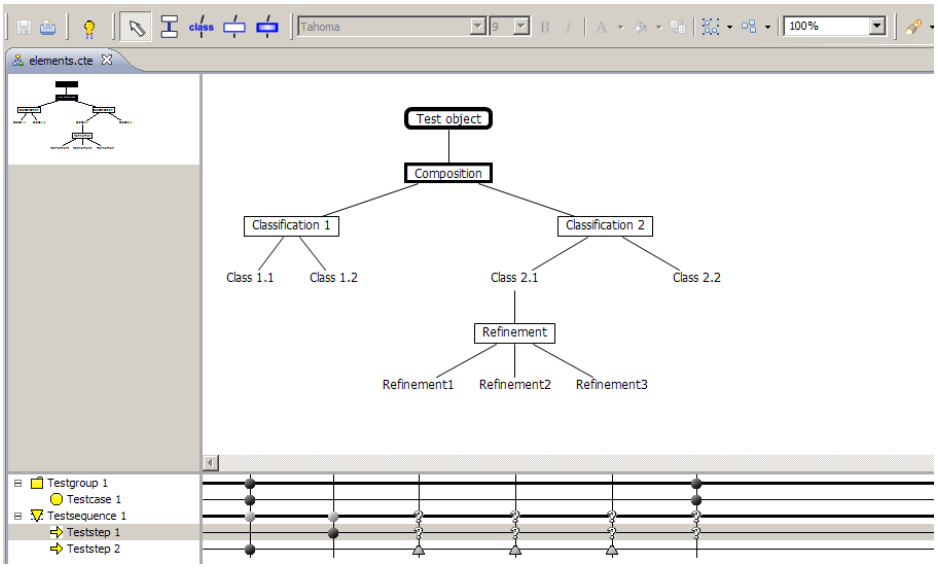

Figure 2.2: CTE XL Layout

The Classification Tree Editor is a graphical editor to create and maintain classification trees [WG93]. It supports the two steps of the classification tree method with distinct areas in its program window. First, the upper part of the program window, the tree editor, is used to span classification trees. Then, the lower part of the program window, containing combination table and test case tree, is used to actually specify test cases.

Lehmann and Wegener have extended the editor [LW00] to adopt results from the field of combinatorial interaction testing [NL11], which allows the automatic generation of test suites after the specification of the system under test has been given.

A screenshot of the extended classification tree editor (CTE XL) is provided in Figure 2.2.

The tree editor is located in the center of the application. The graphical representation of classification tree elements is as follows:

- The **root node** has rounded corners.
- **Classifications** have a thin border.
- **Classes** do not have any border.
- **Compositions** have a thick border.

The lower part consists of a test case tree on the left and the combination table on the right.

In the combination table, the tester can select and combine the classes for a test case. Selected classes are represented by a solid cycle. When there is no selected class in a classification yet, all classes are marked with a question mark.

The test case tree is on the left side. It is used to create and edit test cases. Test cases are represented by a cycle and can be combined into test groups. Test groups are represented by a small folder icon. Test groups can contain further test groups.

Test sequences can also be defined in CTE XL. In contrast to atomic test cases, test sequences consist of several test steps. A sequence is represented with a small triangle, containing test steps with a small arrow. When executing a test sequence, it passes, if all test steps successively pass in the order they are defined.

2.4 Dependencies

With the introduction of combinatorial aspects, Lehmann and Wegener also introduced dependency rules for the CTE XL [LW00]. Dependency rules are used to specify constraints between different elements of the classification tree.

An example for this scenario is list processing (Figure 2.3).

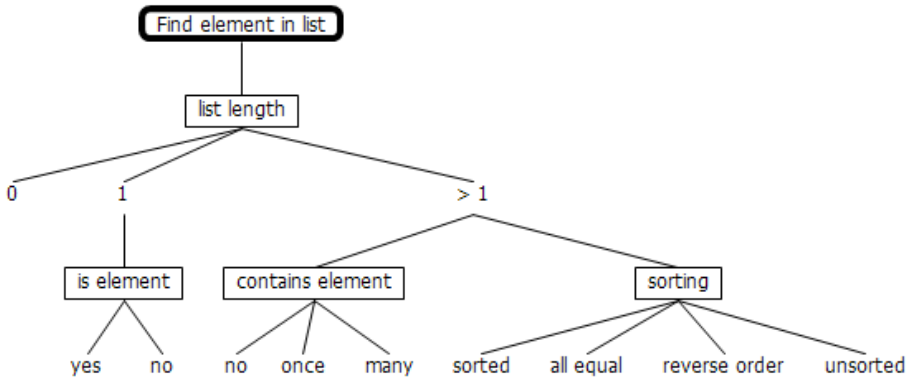

Figure 2.3: Classification Tree for the “Element Counting" Example

The algorithm searches a given list for a given pivot element. The list can be empty (*length* = 0), consist of one element (*length* = 1), or contain more than one element (*length* > 1); resulting in three different list lengths. If the list contains one element

only, it can be either the pivot element or not. With the list consisting of more than one element, the parameter *contains element* can assume the values *no*, *once*, or *many*. The last influence on the algorithm is list sorting. The parameter *sorting* of lists can be *sorted*, *all equal*, *reverse order*, or *unsorted*.

There are several dependencies here. Since *length* > 1 is the only scenario where sorting applies it can be modeled as implicit dependency.

If list sorting is *all equal*, the list can contain the element *several times* or *not at all*, but not just once. This dependency cannot be modeled in the tree, but has to be specified as an explicit dependency for this classification tree.

The CTE dependency manager in Figure 2.4 shows how the dependency between *all equal* and *contains element* with *no* or *many* is formulated in the CTE.

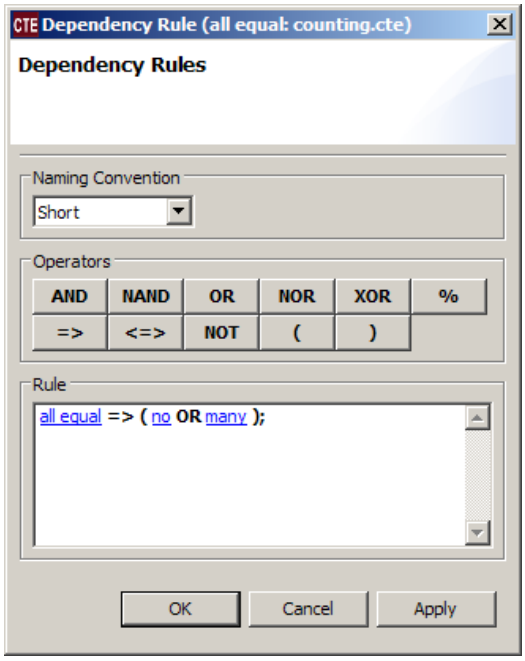

Figure 2.4: Dependency Manager

The CTE XL allows to specify any kind of logical dependency rules, containing {*AND*, *OR*, *NOT*, ⇒, ⇔, *XOR*, *NOR*, *NAND* }. Parentheses are used to formulate more complex expressions.

2.5 Test Case Generation

The original edition of CTE only contained manual test case specification [WG93]. The extended version by Lehman and Wegener introduced automatic test case generation [LW00]. Their CTE XL allows creating test suites that fulfill certain given

coverage levels. Supported coverage levels are *minimal combination*, *pairwise combination*, *threewise combination*, and *complete combination*.

A screenshot of the CTE XL test case generator is provided in Figure 2.5. The minimal combination is represented by the + sign, the complete combination by the * sign. The screenshot shows a generation rule for the list example from Figure 2.1 demanding the complete combination of *list length* and *sorting*.

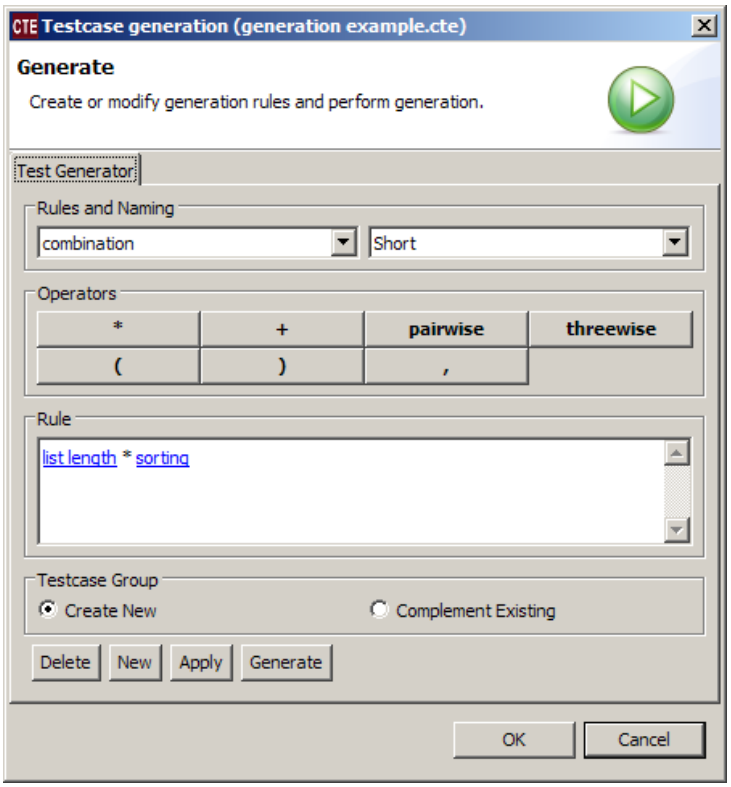

Figure 2.5: Test Case Generator

2.5.1 Test Case Generation with Dependencies

One motivation for this work is the handling of dependencies in CTE XL. Normally, CTE XL handles dependencies quite well and takes care of them during test case generation. There is, however, a problem with concatenated dependency rules: When there are dependency rules which depend on the fulfillment of other dependency rules, CTE XL sometimes tends to not completely handle all dependency rules simultaneously. This behavior results in test suites containing invalid test cases. An invalid test case is a test case violating at least one dependency rule.

A simple example is the classification tree given in Figure 2.6 consisting of four classifications with two classes each. It shall have three dependency rules $d_1 = a \rightarrow !c$,

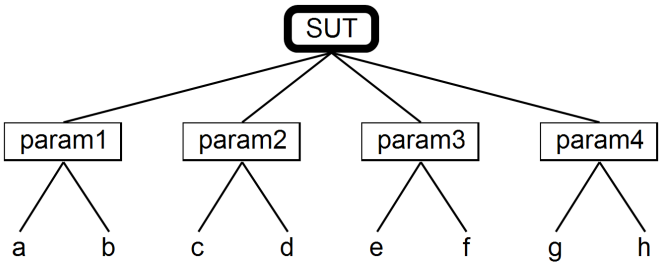

Figure 2.6: Dependency Example

$d_2 = g \rightarrow c$ and $d_3 = e \rightarrow c$. In this simple situation, CTE XL is not capable of generating an all-valid test suite covering all pairs between $param1$, $param2$, $param3$, and $param4$, but always creates a test suite containing invalid test cases due to invalid (forbidden) combinations of classes.

Additionally, the handling of dependency rules resulting in an empty test suite, because all test cases are invalid and thus the number of valid test cases is zero, is not working correctly in CTE XL. For these scenarios, CTE XL still creates test suites, although the tool recognizes, that all containing test cases are invalid afterwards.

2.6 Test Sequence Generation

In [CDFY99] Conrad et al. present the automatic import of Simulink Models into classification trees. They use imported models for systematic determination of test scenarios. Test scenarios can be either test sequences or test cases.

A test scenario is a series of stimuli in a certain order and with assigned durations. They use the combination table of the classification tree editor to define signal courses. With additional metadata, they annotate the type of the course. Supported course types are step, ramp and spline, with step being the default. Their system can be used to model both discrete and continuous signals. The reuse of modeling information from the development for test activities reduces time and costs for test modeling.

The approach is further described in [Con05]. The work focuses on the integration of test scenarios for embedded systems into the development process. Model-based specification, design and implementation are in place, but testing still can be improved. Since testing all feasible combinations is nearly impossible, a good selection of test scenarios determines extent and quality of the whole test. The automatic creation is desirable, but only yet possible to a limited extent, which leads to largely manual test design. The problems of ad-hoc test scenario selections are redundancy and possible gaps. They are typically in a very concrete notation with a low level of abstraction, making reuse difficult.

The given example tree from Figure 2.7 results in the parallel state machine given

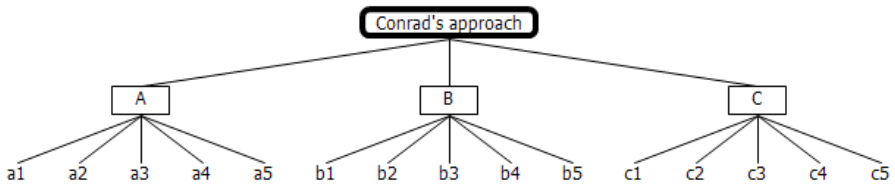

Figure 2.7: Example Tree for Conrad's Approach

in Figure 2.8. The notation and syntax follows [Har87].

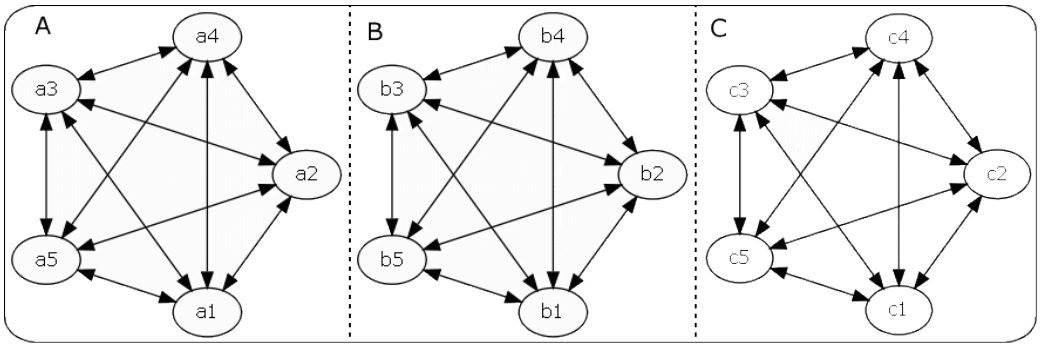

Figure 2.8: Resulting Parallel State Machine

The proposed solution is a model-based testing (MBT) approach [EFW01] based on an abstract model of the input data. The input partitioning of this approach implies a parallel state machine model. Each classification forms one of the parallel parts (AND-states) of the state machine. The states denote the individual equivalence classes defined for the classification. Test sequences can be viewed as paths through such a test model.

Making the underlaying test model explicit allows us to compare Conrad's approach with other MBT approaches. Furthermore, the test model can be used to formalize different coverage criteria. Conrad suggests the systematic scenario selection based on the functional specification.

Current tools supporting the classification tree method do only allow manual definition of test sequences. There are no generation rules for test sequence generation; desired coverage levels for a set of test sequences cannot be specified. Dependency rules to describe constraints between single steps of test sequences do not exist and can therefore not be checked.

3 Related Work

In this chapter, we give an overview of related work. First, we summarize combinatorial and pairwise testing. Then, fundamental work on test sequence generation and validation is described. We then draw conclusions from the related work.

3.1 Combinatorial Testing

Combinatorial Interaction Testing (CIT [CSR06]) is an effective testing approach for detecting failures caused by certain combinations of components or input values. The tester identifies the relevant test aspects and defines corresponding classes. These classes are called *parameters*, their elements are called *values*. We assume the parameters to be disjoint sets. A test case is a set of n values, one for each parameter. In the classification tree method, the parameters are called *classifications*, the values are called *classes*.

CIT is used to determine a smallest possible subset of tests that covers all combinations of values specified by a coverage criterion with at least one test case. A coverage criterion is defined by its strength t that determines the degree of parameter interaction and assumes that all parameters are considered.

The most common coverage criterion is 2-wise (or *pairwise*) testing. It is fulfilled, if all possible pairs of values are covered by at least one test case in the result test set. A large number of CIT approaches have been presented in the past. An overview and classification of approaches can be found in [GOA05] and [KLK08], while [NL11] provide a recent survey of CIT and its evolution. A survey that focuses on CIT with constraints is given in [CDS07]. Nearly all publications investigate pairwise combination methods, but most of them can be extended to arbitrary t-combinations.

The test generation techniques can be classified into *algebraic*, *greedy* and *metaheuristic search* approaches [CDS07, NL11].

3.1.1 Greedy Approaches

AETG

The automatic efficient test generator (AETG) uses a random greedy algorithm. It has been developed at Bellcore and presented by Cohen et al. [CDFP97]. A deterministic variant is also available. Candidates M are generated by selecting the value that is contained in most not yet covered t-tuples. The candidate c is chosen which covers most new t-tuples. A parameter order is created and used to select from each parameter the parameter value with most of the not yet covered t-tuples. AETG supports

pairwise, triple and n-wise. AETG does support seeding and dependencies [CDFP97], but resulting implementation and performance remain unclear. Mixed-strength generation is realized using seeding.

ATGT

The ASM test generation tool (ATGT) uses a logic-based approach [CG10]. The testing problem is mapped onto a model-checking problem. Test predicates are used to formalize combinatorial testing as a logical problem. Then an external formal logic tool is used to solve it. Constraints are expressed as logical predicates as well, which is an advantage over plain tuple exclusion. Constraint processing is done as part of the solving process avoiding expensive pre- or post-processing steps. Test cases are generated as counter examples one at a time while a monitoring process keeps track of covered tuples.

A collecting technique groups not covered tuples to assist in finding good test cases in each step. Final reduction is performed on the complete test suite to avoid redundant test cases. The algorithm is non-deterministic since the tool randomly selects the next predicates for test case generation.

DDA

The deterministic density algorithm (DDA) is an iterative algorithm that generates a test set for pairwise class coverage [CC04].

Each test case is constructed stepwise using a greedy algorithm. Initially, the parameter with the largest *factor density* is selected. The parameter's value is selected by its *level density*.

If there is more than one paramenter with the same level density available, lexicographical order is used as a *tie-breaking rule*. The concepts of densities have the following meaning [BC07]:

- *Factor density* calculates the expected value of not yet covered pairs per parameter.
- *Level density* calculates the expected value of not yet covered pairs per parameter value.

DDA guarantees logarithmic growth of the test suite size in relation to the number of parameters [CC04].

There is a prioritizing variant of DDA which respects the importance of pairs during test case generation [BC06]. The prioritization weights are given by the user. The goal is to cover pairs with high weights early in test case generation. This modification of the DDA for prioritizing test case generation consists of modified density functions. The *weighted density* function calculates the expected value of weights of covered pairs in relation to the number of not yet covered pairs [BC06].

Both variants skip dependencies for performance reasons.

IPO

In parameter order (IPO) was presented by Lei and Tai [LT98]. Their algorithm generates test suites in a constructive way. First a test suite is created to cover only the first two parameters p_1 and p_2. Then all additional parameters are taken into account one after another (p_3 ... p_n). Taking into account additional parameters includes two steps: horizontal growth (the enhancement of existing test cases to have parameter values for newly integrated parameters) and vertical growth (adding new test cases for new combinations (tuples) due to the introduction of parameters). IPO only supports pairwise combination. It does not support dependencies between parameters. The PairTest tool implements IPO.

PICT

Pairwise independent combinatorial testing (PICT) has been developed by Czerwonka. The generation process consists of two phases: preparation and generation [Cze06].

In the preparation phase, all possible parameter interactions are calculated and lists of tuples are composed. Each tuple is marked either *uncovered*, *covered*, or *excluded*.

In the generation phase, test cases are built using a deterministic, heuristic greedy-algorithm. First, seed combinations are added to the current test case candidate as long as they do not violate constraints. From the list with most tuples still uncovered, the first uncovered tuple is chosen. Since tuples have components from different lists (e.g. at least two for $t = 2$), the algorithm iterates through the list(s) of the other part(s) of the tuple. From this second list, tuples are added that do not violate the existing test case configuration. If no new tuple can be taken (because all tuples have already been used in other test cases), an already covered tuple is randomly chosen to be reused. From each list tuples are taken, so that in the end, there is a selection for each list and the test case is complete. Generation finishes when there are no more *uncovered* tuples left.

PICT supports seeding, dependencies and t-wise generation for any t. Mixed-strength criteria and parameter hierarchies are also possible with PICT.

Spec Explorer

Spec Explorer is a CIT generator and a path covering tool [GQW+09]. The solution differs from others as the algorithm generates interaction combination based solely on constraint resolution and model enumeration as provided by the constraint engine. In contrast to other approaches which typically work bottom-up (single test cases are created until coverage is completed), the algorithm creates the result set in a top-down fashion as the solver enumerates combinations. The approach supports constraints and uses heuristics heavily to overcome scalability issues when internally storing the complete set of all possible combinations. It goes one step further than ATGT [CG10] by avoiding reduction steps.

The benchmarks and comparisons by others [SRG11], however, show that performance in terms of result set size needs further improvements. Test generation takes much longer than with the other approaches presented in this section.

3.1.2 Meta-heuristic Search Approaches

CASA

A technique for compiling constraints into a Boolean satisfiability (SAT) problem and integrating constraint checking into existing algorithms is presented in [CDS07]. The technique is integrated into both greedy and simulated annealing algorithms and experiences with its application are reported. The authors provide description for constraint handling, their prototype is proposed to be extendable to other algorithms and to incorporate constraints into algorithms for construction constrained covering arrays.

In the simulated annealing algorithm (as an example for meta-heuristic searches), the outer search is a binary search because result set size is not known at the start. Therefore it needs multiple runs to find a good result set size. In each inner search, the actual annealing takes place. An array is filled with valid values and then gradually improved to become a covering array. Constraints can be given as Boolean formulae allowing SAT solvers to be used for checking. Forbidden tuples are marked as already covered to support the algorithm.

The initial approach had some scaling issues, so major improvements resulted in the final Covering Arrays by Simulated Annealing (CASA) approach [GCD09]. It uses one side narrowing now instead of the binary search and checking of single items has been enhanced to check for slightly larger groups.

3.1.3 Algebraic Approaches

MOLS

Test case generation for pairwise combination can be performed using Mutually Orthogonal Latin Square (MOLS) [MN05]. A Latin square of order m is an $m \times m$ matrix. Each field in the matrix is filled by one of the m different numbers. In each column and each row, each number occurs exactly once.

The first step is to find the MOLSs of a given order. The order of MOLSs is defined by the number of parameter values per parameter. The generation of test cases is then done by reading the details from the MOLS. The algorithm and definitions are given in [MN05]. The problem with MOLS is that their computation is rarely trivial. For prime numbers and the power of prime numbers, there is an algebraic approach. For all other cases, there is no efficient way to create MOLS. If all parameters have the same number of parameter values, the algorithm creates a reasonable set of test cases. If not, MOLSs are created with maximum order, which is the parameter with the largest number of parameter values. This leads to many redundancies in the resulting test suite. However, there are methods to optimize these test suites [Wil02].

This approach does not support dependencies. Because of their fixed size and order, there is no way to incorporate additional information (e.g. semantics, weights ...) to test case generation.

3.2 Test Sequence Generation and Validation

Model checking aims to prove certain properties of program execution by completely analyzing its finite state model algorithmically [BE10, JM09]. Provided that the mathematically defined properties apply to all possible states of the model, it is proven that the model satisfies the properties. However, when a property is violated somewhere, the model checker tries to provide a *counter-example*. Being the sequence of states, the counter-example leads to the situation which violates the property. A big problem with model checking is the *state explosion problem*: The number of states may grow very quickly when the program becomes more complex, increasing the total number of possible interactions and values. Therefore, an important part of research on model checking is *state space reduction*, to minimize the time required to traverse the entire state space.

The *Partial-Order Reduction* (POR) method is regarded as a successful method for reducing this state space [JM09]. Other methods in use are *symbolic model checking*, where construction of a very large state space is avoided by use of equivalent formulas in propositional logic, and *bounded model checking*, where construction of the state space is limited to a fixed number of steps.

Two temporal logics are compared and debated extensively [Var01], *Linear temporal logic* (LTL) and *Computation Tree Logic* (CTL). Temporal logics describe model properties and can reduce the number of valid paths trough the model.

Heimdahl et al. briefly surveys a number of approaches in which test sequences are generated using model checking techniques [HRV+03]. The common idea is to use the counter-example generation feature of model checkers to produce relevant test sequences.

Krupp and Müller introduce an interesting application of CCTL logic for the verification of manually created test sequences in classification trees [KM05]. Using a real-time model checker, the test sequences and their transitions are verified by combining I/O interval descriptions and CCTL expressions.

Several researchers propose other approaches for test sequence generation. Wimmel et al. [WLPS00] propose a method of generating test sequences using propositional logic.

Ural [Ura92] describes four formal methods for generating test sequences based on a finite-state machine (FSM) description. The question to be answered by these test sequences is whether or not a given system implementation conforms to the FSM model of this system. Test sequences consisting of inputs and their expected outputs are derived from the FSM model of the system, after which the inputs can be fed to the real system implementation. Finally, the outputs of the model and of the implementation are compared.

Bernard et al. [BLLP04] have done an extensive case study on test case generation using a formal specification language called *B*. Using this machine-modeling language, a partial model of the GSM 11-11 specification has been built. After a system of equivalent constraints was derived from this specification, a constraint solver is used to calculate boundary states and test cases.

Binder [Bin99] lists a number of different oracle patterns that can be used for software testing, including the simulation oracle pattern. The simulation oracle pattern is used to simulate a system using only a simplified version of the system implementation. Results of the simulation are then compared to the results of the real system. We can regard the formal model of the system as the simulation of the system from which expected results are derived.

Geist et al. partition a test problem into aspects of interest to guide the search for test cases to interesting parts of the system [GFL+96], using temporal logic and BDDs instead of traditional graph-algorithmic models. The target is transistion coverage. Steps are a) building an FSM model of the test problem, b) definition of coverage model, and c) test generation. All FSM transitions are stored in a BDD for performance reasons. Test cases are generated per transition. New test cases are evaluated for all included transitions and removed from the list of transitions to be covered. The result set size does not necessarily have a minimum number of test cases. The limitation of current coverage tools drives verification by simulation to rely on massive simulations without an inherent way to drive the test generation process by coverage considerations. Geist et al. deem sysbolic exploration not to be a bottleneck. Their technique avoids state-space explosion. Their generation creates many test sequences of medium length, so they propose future work on creation of longer test sequences. They see, however, a tradeoff between reduced simulation time caused by setup conditions with longer test sequences and easier debugging and tracing with shorter test sequences. Therefore they suggest to make the maximum number of transitions per test sequence a user-configurable parameter.

Automatic test sequence generation and coverage criteria for testing of ASMs are discussed by Gargantini and Riccobene [GR01].

Burton et al. present an approach to use formal specification from statecharts and a testing heuristic to automatically generate test cases [BCM01]. For all transitions in the statechart a Z-representation is extracted. The internal Z-representation is then used to create an internal representation. A test sequence is then created for each state of the internal representation. There is no minimization of test sequences.

To generate tests from Z specifications, the disjunctive normal form (DNF) method can be used, although it is prone to state explosion [HHS03]. The authors propose to construct a classification tree from the Z specification and use the resulting tree for test generation. There are several suggestions for constraint learning and efficient tree construction, although the main manual work of test case selection is left to the tester.

Windish has applied search-based testing to Stateflow Statecharts [Win08, Win10].

A messy genetic algorithm (GA) is used to generate transition tours through Simulink

Stateflow models [OHY11]. Oh et al. identify two main challenges: Trigger blocks containing timing constraints or counters and cyclic paths which might require several traversals before triggering a transition. A further problem is the a priori unknown length of the resulting tour. Stateflow models supports hierarchies and concurrences which they directly used to avoid sequentialization and therefore do not suffer from state explosion.

A technique for test sequence generation is introduced by Kuhn et al.: They generate event sequences for a given set of system events. They allow specifying t-way sequences, which includes all t-events being tested in every possible t-way order [KKL10].

Recent application of test sequence generation can be the state-based testing of AJAX web applications [MTR08]. The authors apply model extraction and model learning to AJAX web applications. The resulting FSM is then tested using concrete data from traces obtained during model learning.

3.3 Conclusion

There are only limited possibilities to support explicit prioritization of parameters and parameter values. The only known algorithm supporting prioritized test case generation is the deterministic density algorithm (DDA) published in [BC06], which is an extension of [CC04]. The extended algorithm generates a test suite by successively constructing single test cases. During test case construction it accounts for (1) uncovered pairs in the test suite generated so far and (2) user assigned weights. Pairs with higher weights are covered earlier than pairs with lower weights. For efficiency reasons, this algorithm does not consider explicit dependencies. To our knowledge, it does not support any t-wise combination other than pairwise.

Therefore, we will design new test case generation algorithms for classification trees with incorporated prioritization weights. In [HRSR09] a first approach for combining classification trees with priorities has been presented.

Currently, there is no test sequence generation available for combinatorial testing. The only known approach is the t-way sequence generation [KKL10] which is limited to combinations of parameter values of one parameter at a time and does not handle parameter interactions of several parameters.

For that reason, we will also design test sequence generation algorithms for classification trees.

4 Enhancements

In this chapter, we design our enhancements for test case and test sequence generation with the classification tree method. We select three prioritization models and give a prioritization example to then qualify the classification tree (Section 4.1). For dependency handling, we transform both the classification tree and its dependency rules into an integrated data structure, a Binary Decision Diagram (Section 4.2). Afterwards we use this data structure for dependency handling in prioritized test case generation (Section 4.3). Additionally, we exploit the data structure for a new deterministic test case generation (Section 4.4). For our new test sequence generation we design new dependency and generation rules (Section 4.5). Actual generation is then done by converting the classification tree into a hierarchical concurrent finite state machine and traversing it using a multi-agent system (Section 4.6).

4.1 Prioritization and Qualification

Prioritization is used to allow the assignment of values of importance to several classification tree elements. The values of importance are called weights. To cover all kinds of test aspects, these weights can differ. Higher and lower weights should reflect higher and lower importance, respectively. Consequently, we are able to compare the elements of the classification tree to determine their importance under a given test aspect and to prioritize test aspects during test case generation.

We analyzed several existing prioritization techniques. Elbaum et al. give good overviews of existing approaches [EMR02, ERKM04].

Many existing prioritization techniques are not applicable in the classification tree for methodical reasons. Among these can be that the level of abstraction does not fit or the required details are not included in the classification tree. For example, test cases cannot be optimized on state coverage since the classification tree method is a black-box test design approach.

The following three models have been selected to provide a basis for prioritization as they are applicable for the classification tree method:

- Prioritization based on a **usage model** [WPT95]: This prioritization tries to reflect usage distribution of all classes in terms of usage scenarios. Classes with high occurrence are assigned higher weights than classes with low occurrence.

- Prioritization based on an **error model** [EMR02]: This prioritization aims to reflect distribution of error probabilities of all classes. Classes with high prob-

ability of revealing an error are assigned higher weights than classes with low probability.

- Prioritization based on a **risk model** [Aml00]: This prioritization is similar to prioritization based on error model, but additionally takes error costs into account. Risk is defined as the product of error probability and error costs. Classes with a high risk are assigned higher weights than classes with low risk.

The selected prioritization models will now be used for qualifying the classification tree.

4.1.1 Example

We will use the following example throughout all prioritized generation algorithms. The example in figure 4.1 shows a classification tree for the system under test *Adaptive Cruise Control* (ACC). Its task is to adapt the speed of a vehicle to keep a certain distance to preceding vehicles. Three aspects of interest (*Speed*, *Daylight* and kind of *Preceding Vehicle*) have been identified for the system under test. These classifications are direct children of the root node. The classifications are partitioned into classes which represent the partitioning of the concrete input values. In our example the refinement aspect *Shape* is identified for the class *Car* and it is divided further into the two classes *Limousine* and *Cabriolet*.

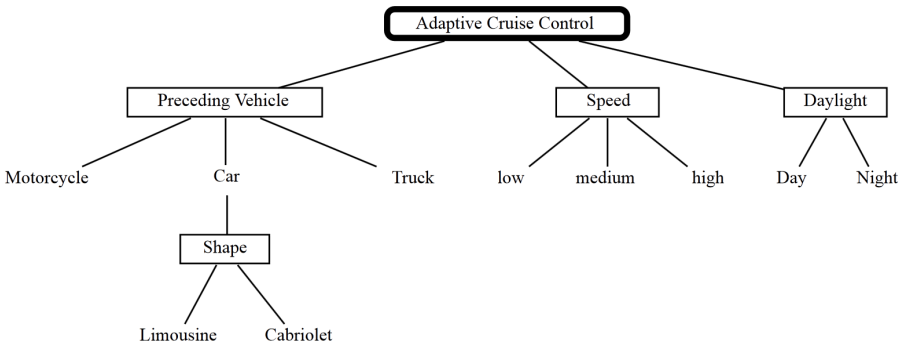

Figure 4.1: Test Object *ACC*

For the qualification with different priority models, there are several things to consider: Consistency is needed both locally (considering a class and all siblings) as well as globally (interaction of classes from different classifications). Additionally, for combined constructs, such as class pairs and other tuples, a uniform calculation is needed. For refined elements, we also need to define a handling. The presence of constraints can have an impact on the priority model as well, so we need to clarify this issue.

4.1.2 Qualification with Usage Model

Occurrence Probability is always given for classes with respect to the parent classification. For class c an occurrence value p_c denotes the relative frequency for c in comparison to all sibling classes d, e, f.... The sum of all occurrence values p_n for siblings from classification C must always be 1. We require that a value is given for all classes of all classifications in the classification tree, before these values can be used for prioritized test generation. Occurrence values for siblings are directly comparable without normalization. Given that classifications are independent of each other, occurrence values from different classifications are also directly comparable. Under this assumption, composed constructs, such as pairs, can be formed by multiplying individual values of all involved classes.

We define refined classes to be interpreted as conditioned probabilities. The actual value can then be calculated using the Bayes-Theorem.

Since handling of constraints is done without knowledge of priorities, the qualification of the classification tree has to be done with constraints in mind.

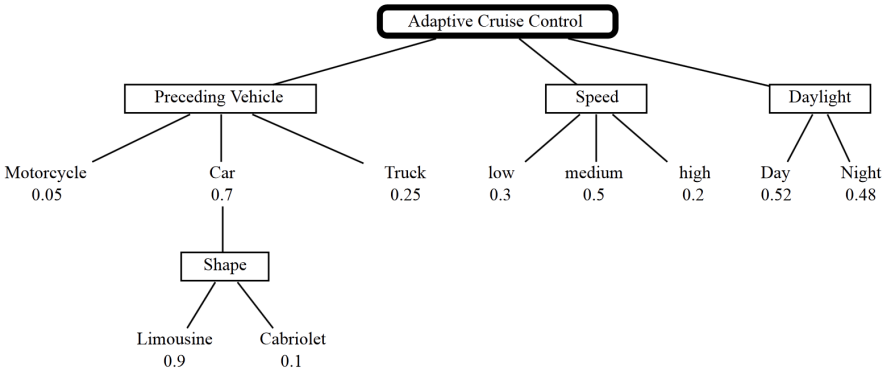

Figure 4.2: *ACC* Test Object with Occurrence Values

Occurrence values for the ACC example are given in Figure 4.2.

Table 4.1: Occurrence Probabilities for Class Pairs

		low	medium	high	Day	Night
		0.3	0.5	0.2	0.52	0.48
Motorcycle	0.05	0.015	0.025	0.01	0.026	0.024
Limousine	0.63	0.189	0.315	0.126	0.3276	0.3024
Cabriolet	0.07	0.021	0.035	0.014	0.0364	0.0336
Truck	0.25	0.075	0.125	0.05	0.13	0.12
Day	0.52	0.156	0.26	0.104	-	-
Night	0.48	0.144	0.24	0.096	-	-

All expected occurrence probabilities for combined class pairs are given in Table 4.1.

4.1.3 Qualification with Error Model

Error Probability is always given as absolute values for classes without any relation to the parent classification. For class c an error value e_c denotes the absolute value for c in comparison to all other classes $d, e, f...$ of the classification tree. We require that a value is given for all leaf classes in the classification tree, before these values can be used for prioritized test generation. Error values for siblings are directly comparable without normalization, as well as error values from different classifications. Composed constructs, such as pairs, can be formed by multiplying individual values of all involved classes.

Since error values are independent of each other, we do not introduce a handling for refined classes as conditioned probabilities. Instead, we only allow error values on leaf classes.

Again, handling of constraints is done without knowledge of priorities, so the qualification of the classification tree has to be done with constraints in mind.

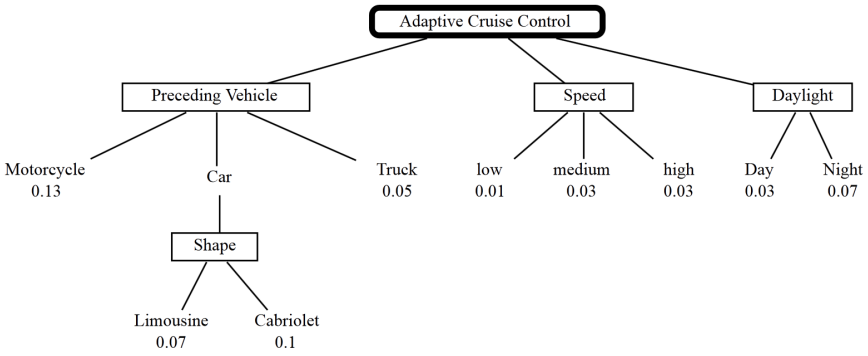

Figure 4.3: *ACC* Test Object with Error Values

Error values for the ACC example are given in Figure 4.3.

Table 4.2: Error Probabilities for Class Pairs

		low	medium	high	Day	Night
		0.01	0.03	0.03	0.03	0.07
Motorcycle	0.13	0.0013	0.0039	0.0039	0.0039	0.0091
Limousine	0.07	0.0007	0.0021	0.0021	0.0021	0.0049
Cabriolet	0.1	0.001	0.003	0.003	0.003	0.007
Truck	0.05	0.0005	0.0015	0.0015	0.0015	0.0035
Day	0.03	0.0003	0.0009	0.0009	-	-
Night	0.07	0.0007	0.0021	0.0021	-	-

All expected error probabilities for combined class pairs are given in Table 4.2.

4.1.4 Qualification with Risk Model

Risk values are always given absolute for classes without any relation to the parent classification. For class c a risk value r_c denotes the absolute value for c in comparison to all other classes d, e, f... of the classification tree. Risk is given in two values, error probability and cost of an error. The actual risk, then, is the product of the two. We require that values are given for all leaf classes in the classification tree, before these values can be used for prioritized test generation. Risk values for siblings are directly comparable without normalization, as well as risk values from different classifications. Composed constructs, such as tuples, can be formed by multiplying the combining individual error probabilities with the summed individual costs.

Since risk values are independent of each other, we do not introduce a handling for refined classes as conditioned probabilities. Instead, we only allow risk values on leaf classes.

The qualification of the classification tree has to be done with constraints in mind, since handling of constraints is done without knowledge of priorities.

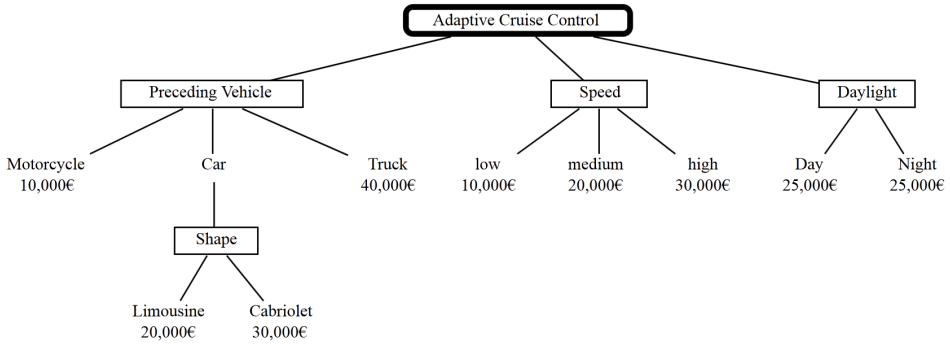

Figure 4.4: *ACC* Test Object with Cost Values

Cost values for the ACC example are given in Figure 4.4. The resulting risk values are given in Figure 4.5.

Table 4.3: Costs for Class Pairs

		low	medium	high	Day	Night
		€10,000	€20,000	€30,000	€25,000	€25,000
Motorcycle	€10,000	€20,000	€30,000	€40,000	€35,000	€35,000
Limousine	€20,000	€30,000	€40,000	€50,000	€45,000	€45,000
Cabriolet	€30,000	€40,000	€50,000	€60,000	€55,000	€55,000
Truck	€40,000	€50,000	€60,000	€70,000	€65,000	€65,000
Day	€25,000	€35,000	€45,000	€55,000	-	-
Night	€25,000	€35,000	€45,000	€55,000	-	-

All expected costs for combined class pairs are given in Table 4.3.
All expected risks for combined class pairs are given in Table 4.4.

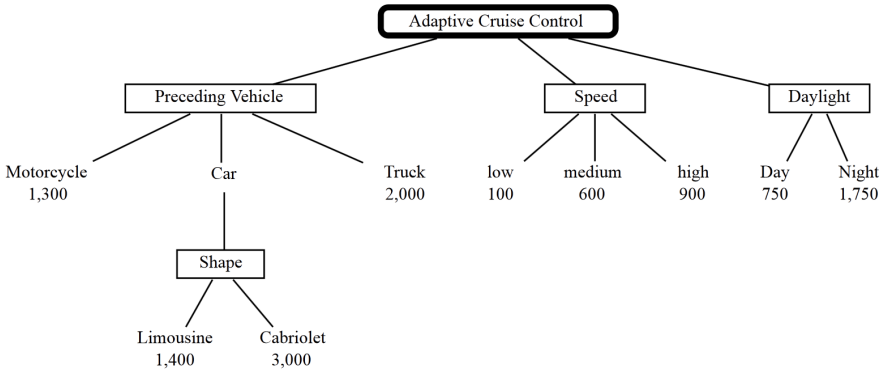

Figure 4.5: *ACC* Test Object with Resulting Risk Values

Table 4.4: Risks for Class Pairs

		low	medium	high	Day	Night
		100	600	900	750	1,750
Motorcycle	1,300	26	117	156	136.5	318.5
Limousine	1,400	21	84	105	94.5	220.5
Cabriolet	3,000	40	150	180	165	385
Truck	2,000	25	90	105	97.5	227.5
Day	750	10.5	40.5	49.5	-	-
Night	1,750	24.5	94.5	115.5	-	-

4.1.5 Conclusion on Qualification

In Table 4.5 we provide additional values for occurrence probability, failure probability, failure costs, and the calculated risk for all leaf classes from our example.

Table 4.5: ACC Test Object Information

	Preceding Vehicle			
	Motorcycle	Limousine	Cabriolet	Truck
Usage	0.05	0.63	0.07	0.25
Failure	0.13	0.07	0.1	0.05
Cost	€10,000	€20,000	€30,000	€40,000
Risk	1,300	1,400	3,000	2,000

	Speed			Daylight	
	low	medium	high	Day	Night
Usage	0.3	0.5	0.2	0.52	0.48
Failure	0.01	0.03	0.03	0.03	0.07
Cost	€10,000	€20,000	€30,000	€25,000	€25,000
Risk	100	600	900	750	1750

Note the difference between the occurrence probability values in the table and the figure for the subclasses of the class *Car*. We explained in previous sections that the occurrence probabilities for classes are conditional probabilities, having a refined class as an ancestor node in the classification tree. In our example the probability of the class *Cabriolet* is 0.1 only when the class *Car* is chosen. Therefore, the absolute occurrence probability for this class is 0.07.

As shown in Table 4.5 it is, for example, much more probable that the preceding vehicle is a *Limousine*. If we focus on finding errors connected to the most common usage scenario, we will expect test cases covering the class *Limousine* to be more important than those covering the class *Cabriolet*. But if we focus on finding errors that have a high risk, we prefer test cases covering the class *Cabriolet* since it is associated with the highest risk.

4.2 Constraints Handling

Classification trees can have implicit dependency rules in terms of refinements. Classes can have ancestor classifications with only local validity [GG93].

In the example from Section 4.1.1, the class *Car* has a refinement in terms of the classification *Shape* (Figure 4.1). The test aspect *Shape* therefore only applies if and only if the *Preceding Vehicle* is a *Car*. If it is a *Truck* or a *Motorcycle*, the classification *Shape* does not apply because *Car* has not been selected.

In contrast to implicit dependency rules in the classification tree method, the classification tree editor CTE XL also allows the specification of explicit dependency rules [LW00].

Explicit dependency rules allow specifying details of the test system, which seem to be orthogonal to the classification tree. It is possible to exclude combinations of

classes from different classifications, for example.

For our example, we will now introduce two dependency rules. First we require, that *Motorcycles* can only occur during the *day*, or more formal:

$$D_1 = Motorcycle \rightarrow Day$$

Additionally we assume, that *Trucks* cannot drive with a *high Speed*:

$$D_2 = Truck \rightarrow \neg high$$

These explicit dependency rules are heavily used by test engineers, but they are problematic during test case generation. In test case generation, there needs to be a check for validity making the whole test generation process a complicated task.

For test case generations, there are three different ways of handling explicit dependency rules: afterwards, during, and beforehand.

Handling after Test Case Generation

The simplest realization is the handling of dependencies just after the actual test case generation. A normal, regular test case generation is performed, while completely ignoring any dependency rules. When the generation is finished, each test case is checked for validity. Invalid test cases are eliminated and new test cases are generated for the now missing tuples, if needed.

One disadvantage of this method is that there might not be any valid test at all containing the missing tuple, so invalid test cases will be generated again and again. The test case selection has no connection to the validity of test cases, checking and filtering is performed afterwards. This approach is only applicable for small test problems, where all possible combinations can be tried.

Immediate Handling during Test Case Generation

The advantage of this approach is the handling of dependency rules during test case generation. This approach will generate only valid test cases, so there is no need for checking and filtering afterwards. This approach, however, requires numerous operations. When composing a single test case, there needs to be a check for validity after each single addition of test elements.

For example, an existing valid test case fragment containing class a_i from classification A needs to be re-evaluated again when class b_j from classification B is about to be added, to check, if a_i and b_j can co-exist.

This approach ensures that only valid test cases are generated. The validity checks, however, are rather expensive; they need time. Combinations of classes are evaluated as long as a valid combination is found, which in the worst case might be the last available combination.

Handling before Actual Test Case Generation

This approach allows the test case generation without taking care of dependency rules, since there are only valid combinations in the classification tree. Invalid combinations do not exist in the tree. There is only one checking and filtering operation before the actual test case generation. There is no need for filtering after the test case generation and there are no expensive comparison operations during test case generation. The actual test case generation therefore is rather cheap, the initial preparation of the tree, however, might need additional effort.

4.2.1 Tree Transformation

We are going to present approaches for tree preparation. We require the following conditions to be met:

- The test case generation is performed on a tree without explicit dependencies.
- A prepared tree must have the same semantics as the original tree.
- Transformations must be deterministic. The transformation of equal trees results in the same result.

We use the mechanism of refinements to transform tree and explicit dependency rules into a classification tree with only implicit dependencies. Refinements are used to model aspects, which only apply for certain classes.

We see two tracks here:

- Transformation of only those tree elements, which are affected by dependency rules.
- Full transformation.

Limited Transformation

The approach is rather simple. For all classifications that are part of dependency rules, we perform the following:

- The first classification in the tree remains.
- To all its classes, copies of the second classification are attached.
- For the classes of these clones, copies of the third classification are created and attached, and so on.

In the end, the tree has the same levels of refinements as it had parallel classifications at the beginning. For all leaf classes, there is only one check for validity.

This tree fulfills all three requirements: There are no additional explicit requirements in the tree. The resulting tree has the same semantics as the original tree. We,

however, need to introduce an interpretation for classification-clones. Their interpretation is trivial, we treat copies the same way as originals. Determinism is gained by providing a fixed transformation order. The effort of transformation is determined by the number of dependency rules. For few rules, the resulting number of leaf classes (and therefore validity checks) is low. For many rules, effort will rise to the full transformation approach, especially if all classifications are part of dependency rules.

Full Transformation

The transformation process is rather trivial. Instead of handling only classifications that are part of dependency rules, we simply take all classifications of the classification tree here.

This also meets our three requirements: There are no additional explicit requirements in the tree. The resulting tree has the same semantics as the original tree. We, again, need to introduce an interpretation for classification-clones, which, again, is trivial, since we treat copies the same way as originals. Determinism is gained by providing a fixed transformation order. The transformation is very laborious, especially for trees with only few dependency rules, as there are many leaf classes to be checked afterwards.

4.2.2 Approach

Arriving here, we decided to convert classification trees into logical expressions for Binary Decision Diagrams (BDD). Binary Decision Diagrams [Lee59] can be used to represent Boolean functions as compact data structures. We use the most common variant of BDDs, reduced ordered binary decision diagrams (ROBDD), and use the term BDD for simplicity. ROBDDs feature variable ordering and graph reduction [Bry86].

We perform a transformation of the classification tree to Binary Decision Diagrams from the root of the tree down to all leafs of the classification tree. For each element in the classification tree, there is a logical equivalent.

Classifications

In any classification C, exactly one class c is selected.

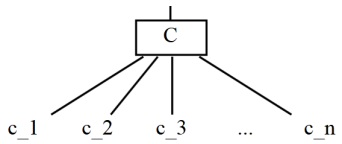

Figure 4.6: Classification C with Classes $(c_1, c_2, \ldots, c_n)$

The classification C with classes $(c_1, c_2, \ldots, c_n)$ (Figure 4.6) results in:

$$\begin{aligned} C = ((&c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \ldots \wedge \neg c_n) \vee \\ &(\neg c_1 \wedge c_2 \wedge \neg c_3 \wedge \ldots \wedge \neg c_n) \vee \\ &\vdots \\ &(\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \ldots \wedge c_n)) \end{aligned} \tag{4.1}$$

Classifications cannot be part of dependency rules, so we can skip their name and structure after conversion.

Compositions

In one composition, there can be classifications and further compositions. For all child elements of compositions there must be selections.

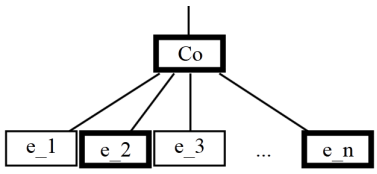

Figure 4.7: Compositions Co with $(e_1, e_2, \ldots, e_n)$

The compositions Co with $(e_1, e_2, \ldots, e_n)$ (with element e_i being composition or classification) (Figure 4.7) results in:

$$Co = (e_1 \wedge e_2 \ldots \wedge e_n) \tag{4.2}$$

Compositions cannot be part of dependency rules, so we can skip their name and structure after conversion.

Classes

Classes can be leaf elements of the tree (Figure 4.8) or they can carry refinement elements (Figure 4.9).

Figure 4.8: Leaf Class Cl

Leaf classes are either selected or not resulting in *true* or *false*:

$$Cl = (Cl)(\textit{for leaf class}) \tag{4.3}$$

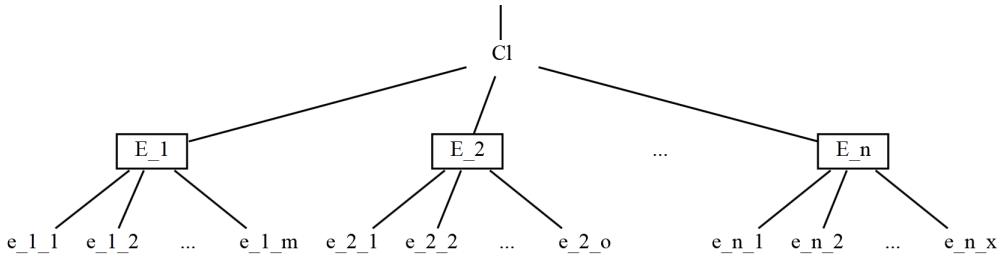

Figure 4.9: Class Cl with Refinements $(E_1, E_2, \ldots, E_n)$

Refined classes can also be *true* or *false*.

If they are *true*, they are treated like compositions. If they are *false*, all sub classes are also *false*.

The class Cl with $(E_1, E_2, \ldots, E_n)$ (with element E_i being composition or classification), results in:

$$\begin{aligned} & \textit{with } Cl = \textit{true} \\ Cl = & (E_1 \wedge E_2 \wedge \ldots \wedge E_n) \\ & \textit{else} \\ Cl = & (\neg e_{1_1} \wedge \neg e_{1_2} \wedge \ldots \wedge \neg e_{1_m} \wedge \\ & \neg e_{2_1} \wedge \neg e_{2_2} \wedge \ldots \wedge \neg e_{2_o} \wedge \\ & \vdots \\ & \neg e_{n_1} \wedge \neg e_{n_2} \wedge \ldots \wedge \neg e_{n_q}) \end{aligned} \tag{4.4}$$

After transformation of the tree, dependency rules $(D_1 \ldots D_n)$ can be attached to the expression E with the $\wedge$-operator:

$$E_D = (E \wedge D_1 \wedge \ldots \wedge D_n) \tag{4.5}$$

This expression now contains all valid test cases. Its construction, however, does not require to calculate or enumerate all assignments.

4.2.3 Example

We will now apply the rules from the previous section and transform the classification tree from Section 4.1.1 (Figure 4.1). The root node is a composition, so using Formula 4.2 the starting Boolean expression is:

$$ACC = PrecedingVehicle \wedge Speed \wedge Daylight \tag{4.6}$$

Using Formula 4.1, we then transform the classifications *Preceding Vehicle*, *Speed* and *Daylight*:

$$\begin{aligned} PrecedingVehicle =& (Mot \wedge \neg Car \wedge \neg Tru) \vee \\ & (\neg Mot \wedge Car \wedge \neg Tru) \vee \\ & (\neg Mot \wedge \neg Car \wedge Tru) \\ Speed =& (low \wedge \neg med \wedge \neg hi) \vee \\ & (\neg low \wedge med \wedge \neg hi) \vee \\ & (\neg low \wedge \neg med \wedge hi) \\ Daylight =& (Day \wedge \neg Nit) \vee (\neg Day \wedge Nit) \end{aligned} \tag{4.7}$$

All classes are leaf classes except class *Car*. For *Car*, we need two variants, a regular and a negated one. Using the first part of Formula 4.4 and Formula 4.1, the regular variant is created:

$$\begin{aligned} Car &= Shape \\ &= (Lim \wedge \neg Cab) \vee (\neg Lim \wedge Cab) \end{aligned} \tag{4.8}$$

Using the second part of Formula 4.4 results in the negated variant:

$$\neg Car = (\neg Lim \wedge \neg Cab) \tag{4.9}$$

Putting it all together:

$$\begin{aligned} ACC =& PrecedingVehicle \wedge Speed \wedge Daylight \\ =& ((Mot \wedge \neg Car \wedge \neg Tru) \vee (\neg Mot \wedge Car \wedge \neg Tru) \vee (\neg Mot \wedge \neg Car \wedge Tru)) \bigwedge \\ & ((low \wedge \neg med \wedge \neg hi) \vee (\neg low \wedge med \wedge \neg hi) \vee (\neg low \wedge \neg med \wedge hi)) \bigwedge \\ & ((Day \wedge \neg Nit) \vee (\neg Day \wedge Nit)) \\ =& ((Mot \wedge (\neg Lim \wedge \neg Cab) \wedge \neg Tru) \vee (\neg Mot \wedge (\neg Lim \wedge \neg Cab) \wedge Tru) \vee \\ & (\neg Mot \wedge ((Lim \wedge \neg Cab) \vee (\neg Lim \wedge Cab)) \wedge \neg Tru)) \bigwedge \\ & ((low \wedge \neg med \wedge \neg hi) \vee (\neg low \wedge med \wedge \neg hi) \vee (\neg low \wedge \neg med \wedge hi)) \bigwedge \\ & ((Day \wedge \neg Nit) \vee (\neg Day \wedge Nit)) \end{aligned} \tag{4.10}$$

As defined in Formula 4.5, the two dependency rules D_1 and D_2 can simply be

attached:

$$\begin{aligned} ACC_D =& ACC \wedge D_1 \wedge D_2 \\ =& ACC \bigwedge(Mot \rightarrow Day) \bigwedge(Tru \rightarrow \neg hi) \\ =& ((Mot \wedge (\neg Lim \wedge \neg Cab) \wedge \neg Tru) \vee (\neg Mot \wedge (\neg Lim \wedge \neg Cab) \wedge Tru) \vee \\ & (\neg Mot \wedge ((Lim \wedge \neg Cab) \vee (\neg Lim \wedge Cab)) \wedge \neg Tru)) \wedge \\ & ((low \wedge \neg med \wedge \neg hi) \vee (\neg low \wedge med \wedge \neg hi) \vee (\neg low \wedge \neg med \wedge hi)) \wedge \\ & ((Day \wedge \neg Nit) \vee (\neg Day \wedge Nit)) \bigwedge(Mot \rightarrow Day) \bigwedge(Tru \rightarrow \neg hi) \end{aligned} \tag{4.11}$$

In the resulting Formula 4.11, each valid assignment of ACC_D now holds a valid test case.

4.3 Prioritized Generation

For the now quantified classification trees, new combination rules need to be designed. These new combination rules respect the weights assigned to the classes in the classification tree. We will introduce the following new prioritizing combination rules: Prioritized minimal combination, prioritized pairwise combination, and class-based statistical combination. For each combination rule, we will define requirements for the resulting test suite. We will then design algorithms for test case generation. Test case coverage criteria will be given where they apply.

4.3.1 Prioritized Minimal Combination

The basic algorithm used to generate minimal combinations creates a test set covering each class at least once. At each step, it generates one test case by selecting single classes from all classifications. It stops generating test cases as soon as all classes are covered, but does not take the order of covering them into account. We extend the algorithm so that it selects classes based on their weights. Classes with a high weight are of great importance for our test objective and will be chosen first. The distribution defined by class weights is used to reselect classes of classification with all its classes already covered by the test suite generated. We compare the empirical class distribution of such a classification with the class distribution given in the generated test suite and choose the class that is most underrepresented in the test suite generated so far.

Coverage Criteria

We define coverage criteria to

- measure to which degree the system under test can be tested with an optimized test suite, and

- to compare optimized test suites of different sizes to determine the benefit of additional test resources.

We define two different criteria that measure the degree to which the classes (each used coverage, EUC) and the weights are covered (weight coverage, WC):

$$\text{EUC} = \frac{\text{number of covered classes}}{\text{number of coverable classes}}$$

$$\text{WC} = \frac{\text{sum of weights of covered classes}}{\text{sum of weights of all coverable classes}}$$

Both metrics are relative, i.e. consider the fact that classes may not be coverable because of dependencies. To understand the need of both criteria we consider a classification with 10 classes where eight have a very small weight. If the test suite consists of only two test cases covering the two classes with the highest weights, we get a small EUC value. But the WC value for all other test suites of size two selecting other classes from this classification will be lower. Using our coverage criteria, the tester can define test end criteria and measure the progress in the test process.

Algorithm

We split the calculation into two algorithms. The top level algorithm calculates the complete test suite. It gets the set M of all valid test cases as input parameter and initializes the set B of all coverable classes extracting them from the set M. Then it iteratively

- passes both sets B and M to the algorithm that calculates the next test case t,
- deletes t from set M and adds it to the result list, and
- deletes all classes covered by t from set B.

The algorithm stops as soon as all classes are covered, i.e. when set B is empty. A class c is only deleted from B, if a test case containing c has been added to the result list. The test cases are selected from the set of valid test cases, i.e. we add only valid test cases to the result list. The algorithm does not consider uncoverable classes since uncoverable classes do not appear in the set B.

The algorithm for defining a single test case t (Figure 4.10) gets at each call a copy of the updated set B of all classes not yet covered and a copy of the updated set M of all valid test cases still available. It reduces M so that it finally contains only one test case. In each iteration, the class k with the highest weight of all still available classes is selected from B (Line 5), taking only already selected classifications into account. This k is used to further restrict M (Line 7). If the restriction with k fails, i.e. there is no test case left in M for the so far selected classes including k, the algorithms backtracks (Line 10) and restarts selecting a new class. If a classification is already covered, there will be no k. In this case, all test cases left in M are compared and the test case that contains the most underrepresented classes is chosen (Line 14).

```
1: M //set of available test cases
2: B //set of classes not yet covered
3: while (|M| > 1) do
4:    N = copy M for backtracking
5:    k = select class from B
6:    if (k was found) then
7:       M = filter M with k
8:       B = delete k from B
9:       if (M is empty) then
10:         M = N //backtracking
11:      end if
12:   else
13:      //check underrepresentation
14:      t = choose test case from M
15:      M = {t}
16:   end if
17: end while
18: t = getSingleElement(M)
19: return t
```

Figure 4.10: Test Case Generation Algorithm for Prioritizing Minimal Combination

Every test case returned by *generateNextTestCase* contains the class with the highest weight compared to all classes not yet covered. After n test case generation steps the presented top level algorithm returns a test suite that covers at least the n most important classes. At each generation step as many new classes as possible are selected, i.e. coverage and weight of classes are maximized.

Example

In Table 4.6 the test suite generated by the presented algorithm is shown for our example using the risk value annotations from Table 4.5 and without any user-defined dependencies. The test suite covers all classes at least once. The classes *Cabriolet, high* and *Night* are covered by the first test case since these classes are associated with the highest risk values. In the last two test cases the class *Night* is reselected for the classification *Daylight*. The algorithm did not choose class *Day* since its risk value is much lower than the value for class *Night*. Since we want to detect failures with a high risk we prefer test cases that cover classes with high risk values.

Table 4.6: Resulting PMC Test Suite for the Risk Model

	Prec. Vehicle	Speed	Daylight	EUC	WC
#1	Cabriolet	high	Night	0.3	0.48
#2	Truck	medium	Day	0.6	0.76
#3	Limousine	low	Night	0.89	0.8
#4	Motorcycle	high	Night	1	1

The last two columns in Table 4.6 contain the coverage values for the test suite containing only the test cases up to that row. We can see that we cover almost half

of the risk values sum with only one test case. Comparing both values EUC and WC helps finding the best test suite size with respect to test resources.

Discussion

The prioritized minimal combination allows adapting the test suite to available resources. In an optimized test suite, classes with high weight are covered and the global coverage of new classes and weights is maximized. If there are no new classes to cover under one classification, a test suite is composed based on the distribution of weights. Whenever there are less classes under one classification than the number of total existing test cases exist, classes with higher weights reoccur in the test suite. Without optimization, a test suite created with the prioritizing minimal combination contains all classes of the classification tree, if there are no dependency rules.

Dividing the number of covered classes by the number of coverable classes gives a good criterion for absolute coverage. Dividing the sum of all weights of covered classes by the sum of all weights of coverable classes gives a good criterion for relative coverage. Due to coverage criteria all results are measurable.

4.3.2 Prioritized Pairwise Combination

The basic pairwise algorithm iteratively generates test cases until all possible class pairs are covered by at least one test case. At each generation step, it tries to select as many new pairs as possible for the test case, but does not take any order of covering them into account. We extend the algorithm so that test cases covering pairs of classes with high weights are generated before those containing pairs of classes with smaller weights.

Coverage Criteria

We define two different criteria, in analogy to the PMC coverage criteria, that measure to which degree the class pairs (each used coverage, EUC) and to which degree the weights are covered (weight coverage, WC):

$$\text{EUC} = \frac{\text{number of covered class pairs}}{\text{number of coverable class pairs}}$$

$$\text{WC} = \frac{\text{sum of weights of covered class pairs}}{\text{sum of weights of all coverable class pairs}}$$

Both metrics are again relative.

Algorithm

Our prioritization pairwise combination algorithm is based on pair weights that we define for the different prioritization models. For the usage and the failure model we define the weight of a pair (c_i, c_j) as the product of the weights assigned to c_i and c_j.

```
1: S //result list
2: M //set of all valid test cases
3: A //set of test cases containing pair p
4: P //set of not yet covered class pairs
5: D //set of (test case, index value) pairs
6: while (|P| > 0) do
7:    p = select max weight pair from P
8:    A = filter M by p
9:    if |A| > 1 then
10:      D = calculateIndex(A, P)
11:      t = selectMaxIndex(D)
12:   else
13:      t = take single test case from M
14:   end if
15:   S = append(S, t)
16:   M = M − {t}
17:   P = P − {classPairs(t)}
18: end while
19: return S
```

Figure 4.11: Test Suite Generation Algorithm for Prioritizing Pairwise Combination

The risk weight is calculated as the product of both failure rates and the sum of c_i and c_j's failure costs. The pair weight values are calculated for all possible class pairs before test suite generation.

The new algorithm selects pairs with a high weight first but at the same time tries to cover as many new pairs as possible. For that reason the algorithm calculates index values that rate the test cases in terms of the covered weights and the number of pairs not yet covered for all suitable test cases. The index value of a test case is the sum of the weights of newly covered pairs and the quotient of the number of newly covered pairs and the number of coverable pairs in a test case. Since we do not use the absolute number of new covered pairs in the index calculation, test cases covering pairs with high weights get higher index values and will be preferred during test suite generation.

Input parameter of the algorithm (Figure 4.11) is the set M of all valid test cases. The algorithm starts initializing the set P with all coverable pairs. Then test cases are generated iteratively. In each iteration, the algorithm first selects the pair p with the highest weight from P (Line 7). To set A it adds all still available test cases from M that contain the pair p. For all test cases in set A the index value is calculated (Line 10) and the case t with the highest index value is chosen and appended to the result list. From several test cases with the same maximum index value, we take the first one.

The selected test case t is deleted from the set of available test cases M and all pairs covered by t are deleted from the set P (Line 17). The algorithm iterates until all pairs are covered by at least one test case, i.e. the set P is empty.

In contrast to the prioritizing minimal combination algorithm, the underrepresentation of already covered class pairs is not taken into account for reselection of class

pairs since this calculation would be very expensive. In each iteration at least the class pair p with the highest weight of all still available class pairs is selected from P. After n generation steps the test suite covers at least the n most important class pairs. At each generation step as many new class pairs as possible are selected, i.e. coverage and weight of class pairs are maximized.

Example

For our ACC example (Figure 4.3) the prioritizing pairwise combination algorithm generates 12 test cases using the failure model (Table 4.7). The first test case covers the pair *(Motorcycle, Night)* since it has the highest weight.

Table 4.7: Resulting PPC Test Suite for Error Model

	Prec. Vehicle	Speed	Daylight	EUC	WC
#1	Motorcycle	medium	Night	0.12	0.23
#2	Cabriolet	high	Night	0.23	0.41
#3	Motorcycle	high	Day	0.35	0.54
#4	Limousine	low	Night	0.46	0.63
#5	Cabriolet	medium	Day	0.58	0.74
#6	Truck	high	Night	0.65	0.81
#7	Limousine	medium	Day	0.73	0.88
#8	Limousine	high	Night	0.77	0.91
#9	Truck	medium	Day	0.85	0.95
#10	Motorcycle	low	Day	0.92	0.98
#11	Cabriolet	low	Night	0.96	0.99
#12	Truck	low	Night	1	1

The last two columns in Table 4.7 contain the values for the coverage criterion. They show that the first three test cases (25% out of 12) from the test suite already cover more than 50% of all class pairs' weight. For covering 50% out of all class pairs, the first five test cases are necessary, which are about 42% of all test cases. For a class pair weight coverage of 90%, 95% or 99%, only eight, nine or eleven tests need to be executed, which are 67%, 75% and 92% of all test cases, respectively. Depending on a tester's needs, a reduction of test effort can be gained.

Discussion

The prioritized pairwise combination allows adapting the test suite to available resources. In an optimized test suite, class pairs with high weight are covered and the global coverage of new class pairs and weights is maximized.

Without optimization, a test suite created with the prioritized pairwise combination contains all class pairs of the classification tree, if there are no dependency rules.

Dividing the number of covered class pairs by the number of coverable class pairs gives a good criterion for absolute coverage. Dividing the sum of all weights of covered class pairs by the sum of all weights of coverable class pairs gives a good criterion for relative coverage. Due to coverage criteria all results are measurable.

4.3.3 Plain Pairwise Sorting

In addition to PPC, we apply a sorting approach based on class pair weights to the results of the plain pairwise algorithm (PPS). The calculation of the weights is same as above. The sorting brings all test cases into an order so that the weight covered by the first test cases is maximized. The algorithm sorts all test cases by their absolute weight at first. Then, it applies as many discriminatory reorderings as there are test cases.

Please note that this approach does not guarantee coverage of any n most important class pairs by the n first test cases. However, the generated test suite will have exactly the same size as the plain pairwise combination as the suite does not grow by sorting. The generation process using sorting is deterministic too; its results, however, differ from the PPC results.

```
1: init (List of all classes)
2: Map of (Pair, double) classPairWeights
3: for all class pair in List of all classes do
4:     combined = weight of first pair tuple * weight of second pair tuple
5:     classPairWeights.put(class pair, combined)
6: end for
```

Figure 4.12: Sort Algorithm Initialization

Figure 4.13 and Figure 4.14 provide the algorithms in pseudocode.

```
1: insert(List of testCases, testCase t)
2: double tcWeight = 0.0
3: /* Calculate insertion weight */
4: for all class pair in t  do
5:     tcWeight += classPairWeights.get(class pair)
6: end for
7: /* Find position for insertion */
8: int i = 0
9: for all testCaseWeights do
10:     if (testCaseWeight < tcWeight) then
11:         testCaseWeights.add(i, tcWeight)
12:         testCases.add(i, t)
13:         return
14:     end if
15:     i++
16: end for
17: /* If all existing test cases have higher weights, append the new test case at the end. */
18: testCaseWeights.add(tcWeight)
19: testCases.add(t)
```

Figure 4.13: Sort Algorithm Insertion

The insertion algorithm requires an initialization (Figure 4.12). For each class pair, the combined weight needs to be calculated by multiplication. A HashMap is filled with the class pair as the keys and the combined weights as the values.

```
1: finalize(List of testCases)
2: if (|testCases| < 2) then
3:     return
4: end if
5: testCase firstElem = testCases.getFirst
6: /* Set weight of all covered pairs to 0 */
7: for all class pair in firstElem do
8:     classPairWeights.put(class pair, 0)
9: end for
10: List of testcases tail = l.subList(1, |testCases|)
11: /* Sort tail using insertion sort */
12: List of testcases newTail = new List
13: testCaseWeights.clear
14: for all testcase in tail do
15:     insert(newTail,testcase)
16: end for
17: /* Finalize tail */
18: finalize(newTail) // recursion
19: testCases = firstElement + newTail
```

Figure 4.14: Sort Algorithm Finalization

In Table 4.8 the expected results for PPC are shown for the ACC example using the usage model (Figure 4.2).

Table 4.8: Resulting Sorting Test Suite for Usage Model

	Prec. Vehicle	Speed	Daylight	WC
#1	Limousine	medium	Night	0.31
#2	Limousine	high	Day	0.52
#3	Truck	medium	Day	0.66
#4	Limousine	low	Night	0.77
#5	Truck	high	Night	0.85
#6	Motorcycle	low	Day	0.9
#7	Cabriolet	medium	Day	0.93
#8	Cabriolet	low	Night	0.96
#9	Truck	low	Night	0.98
#10	Motorcycle	medium	Night	0.99
#11	Cabriolet	high	Night	0.99
#12	Motorcycle	high	Night	1

4.3.4 Class-based Statistical Combination

Class-based statistical combination enables statistical testing using the classification tree method. It is based on the class distribution, given by the user-defined weights, within classifications. The classes of greatest importance for the defined test objective (classes with a high weight) have a greater probability to be chosen for a test case. The goal is to generate a test suite of size n that reflects the defined priority model well.

The test case generation is based on a random process. It might define redundant test cases, i.e. test cases that already exist in the test set. We allow the generation of redundant test cases in order to test non-deterministic systems. In such systems the repeated execution of the same test case increases the probability of revealing failures. The test case generation might also define invalid test cases, i.e. test cases not fulfilling the given explicit dependencies, but these test cases are not added to the test suite.

Using class-based statistical combination, the test suite quality can differ every time we generate a new test suite. This is due to the fact that the suite is the result of a random process. For this reason, we have introduced a method to measure the quality of the test suite. The quality assessment is based on the *chi-square* test [Jan05], a statistical inference procedure. This test allows us to judge whether the class distribution in the generated test suite differs significantly from the distribution of classes that is assumed by the priority model. The *chi-square* test provides us with a value of the p-coefficient which defines the level at which the test suite is significant. Generally, the tester is free to define a significance level that is appropriate for the test domain. A commonly used significance level is $\alpha = 0.05$. If the p-coefficient is less than or equal α, it can be assumed that the test suite reflects the defined priority model sufficiently. If it is not, the test suite can be generated again to achieve a better result.

Algorithm

As input the generation algorithm (Figure 4.15) is given the number of test cases to be generated w and the number of generation steps r. At each generation step it constructs a test case, checks it for validity and, if valid, adds it to the result set. The test case is generated randomly, using the distribution of classes within their classifications. For every classification to be covered in the test case, a class is selected on the basis of a predefined probability (Line 10). Since this construction can produce class combinations that are not allowed, we have to check the validity of the test case (Line 12). We need the parameter r to guarantee termination. For complex explicit dependencies our generation procedure could end in an infinite loop generating only invalid test cases. If the result returned by the generation procedure contains less then w test cases, the user can start a new generation process with a higher value for parameter r, enabling more generation steps.

```
1: r //number of generation steps
2: w //number of test cases to be generated
3: S //set of generated test cases
4: C //set of classifications to be covered
5: while (r > 0&&|S| < w) do
6:     r = r − 1
7:     C = initialize classifications to be covered
8:     while C contains classifications do
9:         c = select classification from C
10:        k = select class using distribution of c
11:        t = add class k
12:        if t is valid test case then
13:            S = S + {t}
14:        end if
15:    end while
16: end while
17: return S
```

Figure 4.15: Test Suite Generation Algorithm for Class-Based Statistical Combination

Table 4.9: Resulting CSC Test Suite for Usage Model

	Prec. Vehicle	Speed	Daylight
#1	Limousine	medium	Night
#2	Truck	medium	Day
#3	Cabriolet	medium	Night
#4	Limousine	medium	Night
#5	Limousine	medium	Day
#6	Limousine	low	Night
#7	Motorcycle	low	Day
#8	Limousine	low	Night
#9	Limousine	low	Night
#10	Truck	high	Day
#11	Limousine	low	Day
#12	Limousine	medium	Day

Example

Table 4.9 shows a test suite with 12 test cases generated by the presented algorithm using the usage model. The value of p equals 0.01, i.e. the distribution of classes in classifications is well reflected in the generated test suite. The empirical distribution of classes in the test suite does not differ significantly from the defined theoretical distribution of classes. Table 4.10 shows both distributions. The class *medium*, for example, occurs in 50% of the test cases. The class *Cabriolet*, in contrast, occurs only in one test case because its weight is very low (0.07). The proportion of classes *Night* and *Day* is similar due to similar weights of the two.

Table 4.10: Empirical and Expected Distribution of Classes in the Test Suite

Class	Expected	Empirical
Motorcycle	0.05	0.08
Limousine	0.63	0.66
Cabriolet	0.07	0.08
Truck	0.25	0.16
low	0.3	0.41
medium	0.5	0.5
high	0.2	0.08
Day	0.52	0.5
Night	0.48	0.5

Discussion

Coverage criteria do not apply. Class-based statistical combination generates a test suite based on the distribution of classes. The test suite is generated in a random process, based on the distribution of classes. Logical dependencies are supported. The *Chi-square*-coefficient and the test for significance allow quantitative measurement of test suite quality. The test suite contains redundant test cases. Therefore, the class-based statistical combination is suitable for the testing of non-deterministic systems.

4.4 Deterministic Test Case Generation

We are now using the unified representation of the classification tree with all logical dependency rules from Section 4.2 to perform our own approach for deterministic combinatorial test case generation.

4.4.1 Preparation

In preparation for test case generation, we first initialize the BDD by transforming the classification tree and attaching dependency rules (Figure 4.16). We call this data structure the BDD from now on.

We then identify the tuples to be covered with the resulting test suite, e.g. for pairwise generation, we calculate all pairs of classes from different classifications. The number and size of the tuples is determined by the test generation rule describing the desired coverage level. The user can specify the desired coverage level and the scope of the operation. Mixed strength generation is supported, too. In addition to the global list C of not yet covered tuples, local lists L are created where each list l only holds tuples from the same parameter interactions (e.g. the two classifications involved into pairs).

In Pre-Check mode, we then check each single tuple i for validity by attaching it to a copy of the BDD with $\wedge$. If the BDD still contains valid assignments after attaching the tuple, then there are test cases containing this tuple, so the tuple remains on

```
 1: transform tree into BDD
 2: attach dependency rules to BDD
 3: calculate tuple lists L
 4: add all tuples to list of not-yet covered tuples C
 5: if pre-check mode then
 6:     for all tuple list l from L do
 7:         for all tuple t from l do
 8:             if (BDD ∧ t) has no valid assignment then
 9:                 l = l − t
10:                 C = C − t
11:             end if
12:         end for
13:     end for
14: end if
```

Figure 4.16: Test suite generation algorithm preparation

the *not yet covered* list C. If, however, the BDD does no longer contain any valid assignment after attaching the tuple, then there are no test cases containing this tuple, the tuple is removed from the not yet covered list C and the local list l.

In Post-Check, all tuples remain in the not yet covered list C.

4.4.2 Phase 1

We have prepared a BDD data structure and we have a list of not yet covered tuples. Actual test case generation now begins. While there are valid assignments A for the BDD, we acquire it (Figure 4.17). The assignment a is interpreted as test case and added to the result test suite S. We analyze the assignment a for containing tuples T. With each containing tuple t, we do two things:

- we exclude t from the BDD by attaching it with $\wedge\neg t$ (and not)
- and we remove t from the not yet covered list C.

All tuples already found will not be included in future test cases, because we exclude them from the BDD completely. This ensures that we receive as many test cases with only new tuples as possible.

We can, however, not guarantee that the number of test cases found this way is the highest possible number of test cases with exclusive tuples. This is due to the order in which valid assignments are selected.

We store the number of covered tuples per test case as threshold th for later reference. When there are no more valid assignments in the BDD, we reset it to the initial state of Phase 1 and continue to Phase 2.

4.4.3 Phase 2

We continue with the initially prepared BDD data structure and a list of remaining not yet covered tuples C. As there are no more test cases with exclusively new

```
1: B = copy of BDD
2: while BDD has valid assignment do
3:     a = get next assignment
4:     S = S + a
5:     T = get all tuples from a
6:     th = |T|
7:     for all tuple t from l do
8:         BDD = BDD ∧¬t
9:         C = C − t
10:        L = L − t
11:    end for
12: end while
13: BDD = B // restore copy of BDD
```

Figure 4.17: Test suite generation algorithm for phase 1

tuples, we now try to fit in as many new tuples into a single test case as possible (Figure 4.18).

We select not yet covered tuples from list L and attach them to the BDD with $\wedge$ (Line 7).

To improve this process, we use tuples from the longest local lists L (Line 4), so we prefer tuples from parameter combinations that have not been covered that much yet.

We sort all tuples lists by their length and iterate trough them. We take the largest list (Line 4) and go trough all tuples from it (Line 6). The tuple is then combined with tuples from all other lists (Line 15). In each step, we make a copy of the BDD by attaching the tuple to it using AND (Line 17). If now the BDD does not contain any valid assignments anymore, we cannot combine this tuple with previously selected tuples. In Post-Check, we check this tuple against the original BDD and remove it now if needed (Line 22). If the tuple is the last tuple and no other tuple fits, we continue with the BDD from the previous iteration. Otherwise we take the matching tuple from the current list and continue with the new BDD (Line 20).

After combining all tuples from the largest list with other tuples, the BDD now contains a set of candidates (Line 31). The candidates consist of valid assignments for the BDD for all tuples from the largest list of not yet covered tuples C. While there are candidates W, we order the list by the number of contained new tuples n (Line 40) and drop candidates with less than threshold tuples th. We add the first candidate from the list to the result test suite S (Line 44), remove the tuples from the not yet covered list C and repeat.

We adapt the threshold when no candidate above the threshold has been found (Line 52). The new threshold is the minimum of either the reduced original threshold or the number of tuples contained in the first candidate.

The algorithm stops, when all local lists are empty, either because all tuples are acovered by test cases or because tuples have been removed because they there are no valid test cases.

```
 1: candidates W
 2: while |C| > 0 do
 3:     Sort tuples lists L by |l_n|
 4:     l_max = longest list from L
 5:     B = copy of BDD
 6:     for all tuple t_max from l_max do
 7:         BDD = B ∧ t_max
 8:         if post-check mode then
 9:             if (BDD) has no valid assignment then
10:                 L = L − t_max
11:                 C = C − t_max
12:                 continue
13:             end if
14:         end if
15:         for all list l from L except l_max do
16:             for all tuple t from l do
17:                 tempBDD = BDD ∧t
18:                 if (tempBDD) has valid assignment then
19:                     BDD = tempBDD
20:                     continue
21:                 else
22:                     if post-check mode then
23:                         if (BDD) has no valid assignment then
24:                             l = l − t
25:                             C = C − t
26:                         end if
27:                     end if
28:                 end if
29:             end for
30:         end for
31:         W = W + valid assignments
32:     end for
33:     for all candidate w from W do
34:         if n < th then
35:             W = W − w
36:         end if
37:     end for
38:     m = |W|
39:     while |W| > 0 do
40:         Sort W by number of tuples t
41:         w = remove first element from W
42:         n = number of not yet tuples t in w
43:         if n >= th then
44:             S = S + w
45:             for all tuple t from w do
46:                 C = C − t
47:                 L = L − t
48:             end for
49:         end if
50:     end while
51:     if m = 0 then
52:         th = min (th*0.95, n)
53:     end if
54: end while
```

Figure 4.18: Test suite generation algorithm for phase 2

4.4.4 Example

We will continue using the example from Section 4.1.1. During preparation, a BDD representing the classification tree is calculated (Formula 4.10). In contrast to Section 4.2.3, we skip dependency rules here. We assume the generation rule to be:

$$pairwise(Vehicle, Speed, Daylight); \tag{4.12}$$

The tuple lists L consists of three local lists l with l_1 = *PrecedingVehicle* × *Speed*, l_2 = *PrecedingVehicle* × *Daylight*, and l_3 = *Speed* × *Daylight*.

There are 12 tuples in l_1, eight tuples in l_2 and six tuples in l_3 (Table 4.11), resulting in 26 tuples not yet covered in global list C. Since we do not use dependency rules in our example, no checking for validity is required even in pre-check mode.

Table 4.11: Initial Tuple Lists

l_1	l_2	l_3
(Mot, low)	*(Mot, Day)*	*(low, Day)*
(Mot, med)	*(Mot, Nit)*	*(low, Nit)*
(Mot, hi)	*(Lim, Day)*	*(med, Day)*
(Lim, low)	*(Lim, Nit)*	*(med, Nit)*
(Lim, med)	*(Cab, Day)*	*(hi, Day)*
(Lim, hi)	*(Cab, Nit)*	*(hi, Nit)*
(Cab, low)	*(Tru, Day)*	
(Cab, med)	*(Tru, Nit)*	
(Cab, hi)		
(Tru, low)		
(Tru, med)		
(Tru, hi)		

In Phase 1, the test suite S contains six test cases read from the BDD:

$$\begin{aligned} S = \{&(hi, Cab, Nit), (med, Cab, Day), (hi, Lim, Day), \\ &(med, Lim, Nit), (low, Tru, Nit), (low, Mot, Day)\} \end{aligned} \tag{4.13}$$

Since each test case consists of three tuples (e.g. $a_1 = (hi, Cab, Nit)$ consists of *(Cab, hi)*, *(Cab, Nit)*, and *(hi, Nit)*), the threshold th is set to 3.

Phase 2 starts with the eight tuples not yet covered in C (Table 4.12), with six tuples from l_1 and two tuples from l_2.

The first set of candidates W is then calculated. The longest list with tuples is l_1. All tuples t still in l_1 are combined with tuples from l_2, the only other remaining list with tuples. The first candidate set W consists of:

$$\begin{aligned} W = \{&\{(Mot, med), (Mot, Nit)\}, \{(Mot, hi), (Mot, Nit)\}, \{(Tru, med), (Tru, Day)\}, \\ &\{(Tru, hi), (Tru, Day)\}, \{(Cab, low)\}, \{(Lim, low)\}\} \end{aligned} \tag{4.14}$$

Since all candidates contain less tuples than the threshold th = 3, no candidate is added to the resulting test suite S. Instead, the threshold is reduced to two, because

Table 4.12: Tuple Lists after Phase 1

l_1	l_2	l_3
(Mot, med)	*(Mot, Nit)*	
(Mot, hi)	*(Tru, Day)*	
(Lim, low)		
(Cab, low)		
(Tru, med)		
(Tru, hi)		

it is the minimum of the old threshold reduced (3*0.95) and the number of tuples in the candidates (2).

In the next iteration, the first candidate $w_1 = \{(Mot, med), (Mot, Nit)\}$ matches the threshold and is added to test suite S. The containing tuples $\{(Mot, med), (Mot, Nit)\}$ are removed from the lists l_1, l_2 and C.

The second candidate $w_2 = \{(Mot, hi), (Mot, Nit)\}$ does not match the threshold anymore, because it only contains one new tuple (Mot, hi). The other tuple (Mot, Nit) has just been removed from C.

The third candidate $w_3 = \{(Tru, med), (Tru, Day)\}$ again matches the threshold. It is added to S and the containing tuples are removed from lists l_1, l_2 and C. All other candidates do not match the threshold, so they are dropped. There are now eight test cases in S.

Since candidates were found in this run, the threshold is not adapted and a new set of candidates is generated.

The longest list with tuples is l_1. Since there are no other lists l_i with remaining tuples, there is no tuple combination. The second candidate set W consists of:

$$W = \{\{(Mot, hi)\}, \{(Lim, low)\}, \{(Cab, low)\}, \{(Tru, hi)\}\}. \tag{4.15}$$

Since all candidates contain less tuples than the threshold $th = 2$, no candidate is added to the resulting test suite S. Instead, the threshold is reduced to one, because it is the minimum of the old threshold reduced (2*0.95) and the number of tuples in the candidates (1).

In the next iteration, all candidates W match the threshold. All candidates are then added to the resulting test suite S, which contains 12 test cases when the generation terminates.

$$\begin{aligned} S = \{&(hi, Cab, Nit), (med, Cab, Day), (hi, Lim, Day), \\ &(med, Lim, Nit), (low, Tru, Nit), (low, Mot, Day), \\ &(med, Mot, Nit), (med, Tru, Day), (hi, Mot, Nit), \\ &(lwo, Lim, Nit), (low, Cab, Nit), (hi, Tru, Nit)\} \end{aligned} \tag{4.16}$$

4.4.5 Variation

In this section, we present an approach for better variation of test case generation in Phase 1 and Phase 2.

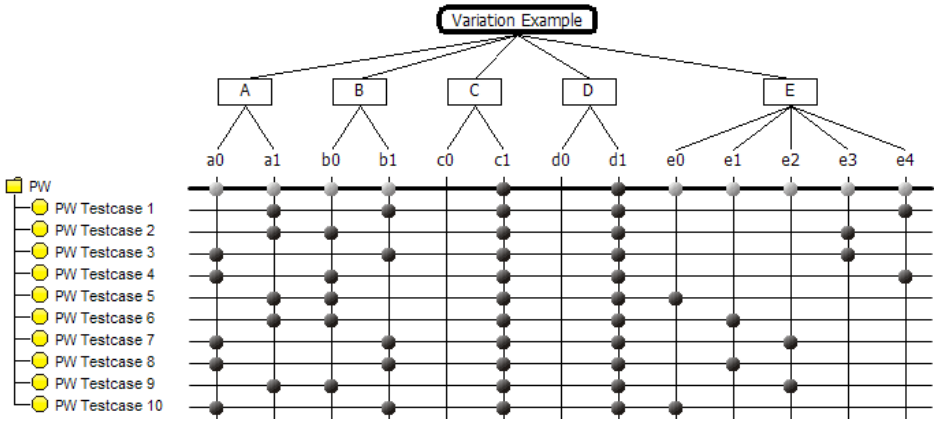

Figure 4.19: Example for Generation without Variation

An example to illustrate the problem is given in Figure 4.19. It shows the expected result for the generation rule $pairwise(A, B, E)$. Test cases 1 to 10 offer a good variation on all classes of classifications references in the combination rule, A, B, and E. For the classifications C and D, there is no variation.

While this result is completely correct, it can be desirable to include some additional variation on those classifications which do not offer new tuples to the test case.

A similar problem occurs when all tuples from a classification have been covered during test case generation and this classification is no longer under consideration for new test cases.

Therefore, we propose the following solution: We apply variation onto all classes of classifications not being part of generation rules and to those classes not part of first time covering tuples.

The algorithm given in Figure 4.20 shows our approach.

```
1: for all selected class without influence on new coverage do
2:     get all (atomic) siblings
3:     select (deterministic) random sibling (can be original class again)
4:     if new testcase is valid then
5:         keep it
6:     else
7:         use original test case
8:     end if
9: end for
```

Figure 4.20: Test Case Variation Algorithm

The algorithm inspects all tuples that do not have any influence on tuple coverage.

This way, there is no need to differentiate between classes of classifications not part of generation rules and classes from classifications already covered. For each class, a list of all class siblings is generated. Only siblings without children are taken into account. It would be possible to include them as well, but refined classes would need to be additionally calculated for selection. From this list of siblings, one class is chosen randomly. To keep the generation process deterministic, random selection uses a fixed initializer. Then, the test case is checked for validity. If it is valid—it still conforms to all dependency rules—it will be used. Otherwise, the original test case will be used. It would be possible to search for another sibling instead of using the original class, but in order to keep calculation time low, the algorithm has only one try.

The final result is given in Figure 4.21. Now all classifications and classes have proper variation.

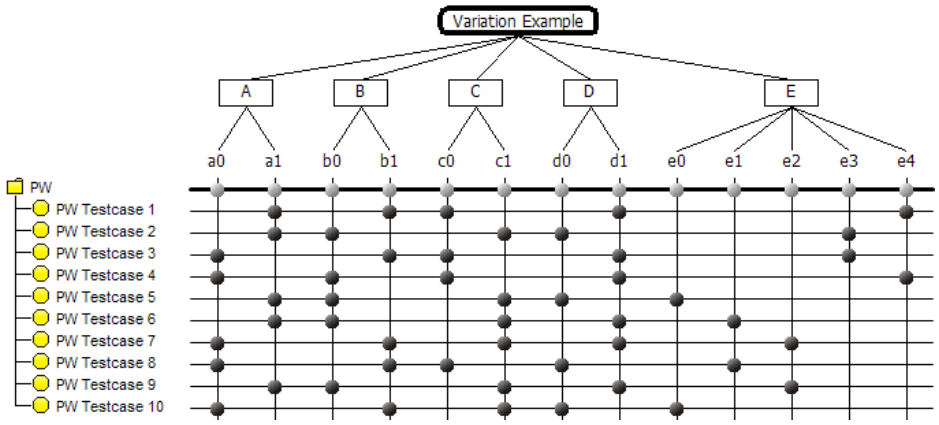

Figure 4.21: Example for Generation with Variation

Future work is needed to analyze, if the result set size can be reduced by covering new tuples with this variation. Additional checking for new tuples would be needed.

4.5 Test Sequence Generation

For our new approach, we want to enable test sequence generation from classification trees. In analogy to existing approaches, we identify three kinds of parameters for test sequence generation: the classification tree itself, dependency rules, and generation rules. The classification tree holds all parameters and their corresponding values of the system under test. For dependency rules, we extend existing rules to new rules describing constraints between single test steps. Our new rules apply per test sequence. Within each test sequence, dependency rules must not be violated. The generation rules describe desired coverage levels for the resulting set of test sequences. The set of test sequences as a whole must respect a single generation rule.

4.5.1 New Dependency Rules

Existing dependency rules allow the user to specify constraints between parameter values of different parameters within one test case. With the new extended dependency rules, it will be possible to specify constraints between parameter values from one test step to another. The following set of dependency rules will be supported (with $i,j,k,n,o \in \mathbb{N}; m \in \mathbb{Z}$):

- If class c_i from classification C is selected in test step t_n, then class c_j from classification C must be selected in the succeeding test step t_{n+1}.
- If $C = c_i$ in t_n, then $C = c_j$ in a later t_{n+m}.
- If $C = c_i$ in t_n, then $C = c_j$ in all t_{n+1} to t_{n+m}.
- If $C = c_i$ in t_n, then $C = c_j$ in all t_{n+m} to t_{n+o}.
- Compositions of any (*AND*, *OR*, *NOT*, *NAND*, *NOR*, *XOR*, ...) combination, e.g. if $C = c_i$ *OR* $B = b_k$ in t_n, then $D =$ *NOT* d_j in a later t_{n+m}.

The existing dependency rules are a subset of our new dependency rules for t_n and t_{n+m} with $m = 0$.

Classic dependency rules are valid for manually created test cases, too. We want our new dependency rules to be available for manually created test sequences, as well.

4.5.2 New Generation Rules

Existing generation rules specify the coverage level of the resulting test suites. For our new generation rules, we want to specify the following parameters:

- Desired coverage level.
- (Sub-)Set of all parameter-value combinations of the classification tree.
- Minimum and maximum number of test steps per test sequence.
- Set of special starting and ending parameter value combinations.
- Maximum number of test step repetitions (both local and global per resulting test sequence).
- Maximum number of parameter value repetitions (both local and global per resulting test sequence).

4.5.3 General Approach

Ideally, the workflow should be as blackbox as possible. After a user has specified the classification tree and all kinds of rules, the approach should work autonomically. We do, however, allow user interaction for advanced users to fine tune the results.

The input for test sequence generation consists of a common classification tree, a set of dependency rules and a single generation rule. Together, they are used to generate an internal representation of the sequence generation problem. Then, a set of valid transitions and steps from one state in the internal representation to the next one is defined, using the new dependency rules. This results in a set of all valid paths through the internal representation. From this set some valid paths are selected to create subsets using the new generation rules. The output is then created by using all remaining sequences.

This obviously leads to numerous sequences.

Even for classical test case generation, it becomes crucial to select subsets from the large set of possible test cases due to test case explosion. For classical test case generation, this can be done by both dependency rules and generation rules.

We identified decision trees and finite state machines as possible internal representations.

We decided to implement the new dependency rules using *linear temporal logic* (LTL) with the operators (X, G, F, U, R). LTL-terms fit well and fulfill most of the requirements defined in Section 4.5.1.

For generation rules, we used *computation tree logic* (CTL). CTL rules do not apply to single sequences but refer to the whole set of generated test sequences.

Additionally, we need some approach-specific rules. In state machines, we want to limit the number of cycles. For decision trees, this setting is not needed. All other needs in generation rules not addressed by CTL will be called approach-specific rules.

The set of valid states consists of the set of all valid test cases, which can already be very large. Therefore, the user starts with the generation of classical test cases and can use them as a subset of all valid test cases for the remaining test sequence generation.

The transitions of all states are then restricted by the dependency rules, generation rules and approach-specific rules. If no dependency rules are given, all transitions are valid in our approach. A transition remains valid as long as there is no contradicting dependency rule. One could, however, define the opposite way: Only transitions specified in dependency rules are allowed whereas transitions not mentioned in or contradicted by dependency rules could be removed.

After removing all forbidden transitions, a set of valid passes through the internal representation remains. By applying generation rules and approach-specific rules, we now select a subset. The approach-specific rules can, for example, limit the path length, the number of cycles or the desired coverage level.

The output is a set of test sequences. Together, they fulfill the single generation rule and approach-specific rules, either one fulfills the dependency rules.

We have initially implemented the generation of test sequences using decision trees and finite state machines. Both approaches follow a simple work flow.

Generation input includes the classification tree, a set of dependency rules and generation rules. For all approaches under evaluation, these three components should be generic. A test suite of test sequence is the output of the generation of test sequences. The actual test sequence generation can be divided into four steps: The derivation of nodes, the definition of valid transitions between these nodes, the composition of valid paths (containing these transitions), and the selection of a subset of all possible paths. The actual generation might need additional *approach-specific* rules.

4.5.4 Decision Tree Approach

For the decision tree approach, the work flow is as follows: The nodes contain all possible combinations of classes from the classifications. The number of nodes equals the size of the complete coverage, the Cartesian product of all classes from all classifications of the tree. Only valid nodes—containing valid class combinations—are added to this set. In this step, all conventional dependency rules and implicit dependencies are handled. For this set, all valid transitions are generated. The dependency rules are applied, where possible. Only those rules describing constraints between consecutive test steps can be processed. From this set of simple transitions, between two nodes each, a decision tree is then composed. Here, all still unprocessed dependency rules are taken into account. The resulting decision tree now contains both all valid paths and only valid paths. In the last step, a path selection is made to fulfill the generation rule. We developed two approaches here, a brute-force and a random approach. The brute-force approach starts traversing the decision tree in a depth first search. When reaching an atomic node in the tree, the path is added to the result set. When reaching a node, which will not reach anything new, this path is skipped. The random approach randomly selects complete paths from the decision tree. Both approaches stop as soon as the generation rule is complete.

4.5.5 FSM Approach

For a given classification tree, the user must specify a set of valid LTL rules. The rules are then used to build a Büchi automaton using *Formula Rewriting*, *Core Translation* and actual *Automaton Generation* following an algorithm [GPV+95] which uses a Tableaux Construction Method for LTL given in [CGP99, p. 132]. The Büchi automaton is then reduced to a regular FSM by removing all infinite properties using weak Next- (and Until-) Operators in Finite Trance LTL and excluding empty sequences. The FSM is then converted into a Test Case FSM by identifying possible test case candidates for each state and removing states without valid test cases. The Test Case FSM can then be traversed with different coverage levels: State Coverage (C0), Transition Coverage (C1) and Path Coverage (C2).

The search for a minimal State Coverage is equivalent to the search for a Hamilton path in a directed graph [Sed03, p. 60]. The search for a minimal Transition Coverage

is equivalent to an asymmetric Traveling Salesman Problem [Pun04]. Both searches are known to be NP-complete. The search for a minimal Path Coverage is equivalent to the search for an Euler path [Sed03, p. 62] or in some cases the Chinese Postman Problem [Sed03, p. 224].

Only Path Coverage has been implemented, a depth first search uses an Adjacency matrix that allows specifying the minimum and maximum search depth (equivalent to the resulting path length) and the minimum and maximum number of state repetitions.

4.6 Statechart Approach for Test Sequence Generation

We will first introduce the refined requirements for the generation rules and then introduce the actual approach.

It can be desirable to have specific coverage levels and it can be useful to generate a test suite with test sequences covering all possible transitions between classes of the classification tree.

4.6.1 New Generation Rules

In analogy to conventional test generation, covering all pairs of transitions between classes of the classification tree could be defined as well. Conventional test case generation supports mixed strength generation as well as seeding [CDFP97], so we require them as well.

New generation rules should allow any t-wise coverage for both classes and transitions. Note that some of Kuhn's t-way sequences [KKL10] can be mapped onto our generation rules.

Kuhn's *1*-way sequence coverage corresponds to *1*-wise (or minimal) class coverage here. Each class is supposed to be contained in the result set at least once. Our approach extends conventional class coverage for test cases to test sequences.

Kuhn's *2*-way sequence coverage corresponds to our *1*-wise (or minimal) transition coverage. All valid transitions (pairs of states) are supposed to be contained in the result set at least once. In conventional test case generation, there is no coverage criterion for transitions.

Higher n-way (with $n > 2$) sequence coverage is not yet included and requires future work. Instead, we require higher n-wise (with $n > 1$) coverage for both classes and transitions. We have included pairwise class and transition coverage.

The generation rules should take classifications as parameters to specify their focus. The number of parameters should not be restricted. Elements of generation rules should be combinable to allow mixed strength generation. Seeding [CDFP97] of manually created test sequences should be possible. The generation algorithm should then analyze this set and take these sequences into account. The generation should, of course, take the dependency rules into account.

4.6.2 Approach

Our final approach for test sequence generation is based on an idea proposed by Conrad [Con05], who suggests that the interpretation of classification trees as parallel FSMs together with a set of test sequences allows measuring coverage levels.

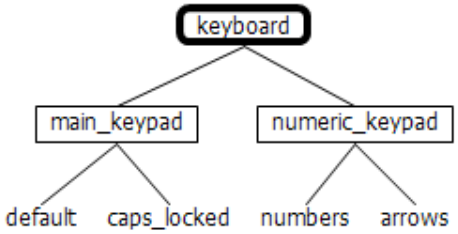

Figure 4.22: Classification Tree for the Keyboard Example

As an example, we use keyboard states: Given a classification tree *keyboard* (Figure 4.22) together with a set of (manually specified) test sequences, we can derive a parallel state machine (Figure 4.23).

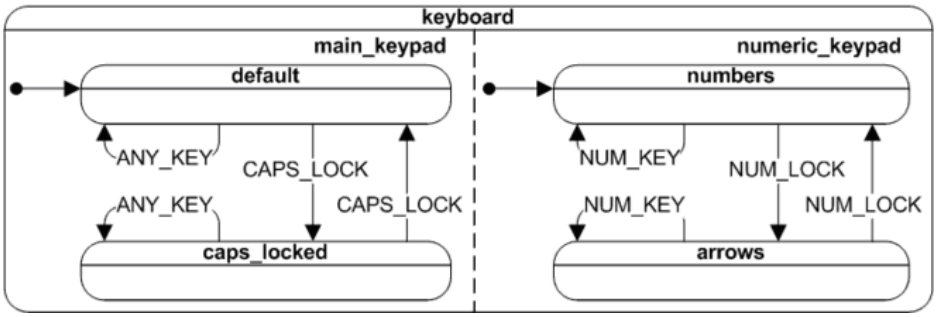

Figure 4.23: Parallel FSM for the Keyboard Example [Mir09]

In UML state charts, parallel states are called orthogonal regions [Obj10, Section 15.3.10].

Conrad's approach, however, lacks some details:

1. He does not give any advice on how to interpret classification trees with refinements. All trees in his examples are flat trees; there are no refined classes.

2. Although mappings of classification trees onto parallel FSMs exist, there is no general interpretation for any existing (parallel) FSM as valid classification tree.

3. There is no distinction between directions of transitions. All examples given do not differ between transitions to or from a node. Loops are missing as well.

4. His approach does not handle dependencies. The test engineer has to decide on his own, which combination of classes and which order of consecutive test steps are valid.

5. There is (to the best of our knowledge) no automatic test sequence generation in Conrad's approach. The test engineer has to specify all test sequences manually.

We will now handle these short comings one by one:

1. The interpretation of refined classes can be easily accomplished by mapping them on to hierarchical states in state machines. This concept is known from Harel statecharts [Har87] as well. As in classification trees, statecharts can be modeled top-down, from overview to detail, by refining states with a set of substates. This allows different levels of granularity within a single statechart at different hierarchies. Parallel FSMs are sometimes also known as Hierarchical Concurrent finite State Machine (HCSM) [Luc93]. We will from now on call Conrad's approach, the statechart approach.

2. Conversion of existing statecharts to classification trees will be explained in the next section (Section 4.6.3).

3. We will differentiate between different transition directions and will enforce loop transitions (transitions where start-node and end-node are the same), if they exist.

4. For handling dependencies, we will use our own initial dependency rule approach for test sequence generation. To have only one central dependency handling, we will model transition guards using this technique as well.

5. The actual test sequence generation will be given in detail in Sections 4.6.4 and 4.6.5.

4.6.3 Conversion of Existing Statecharts to Classification Trees

To illustrate the problem, we will borrow the example from [Luc93].

The microwave (Figure 4.24) is a good example because it contains concurrency and hierarchies.

We propose the following classification tree (Figure 4.25) as an equivalent representation.

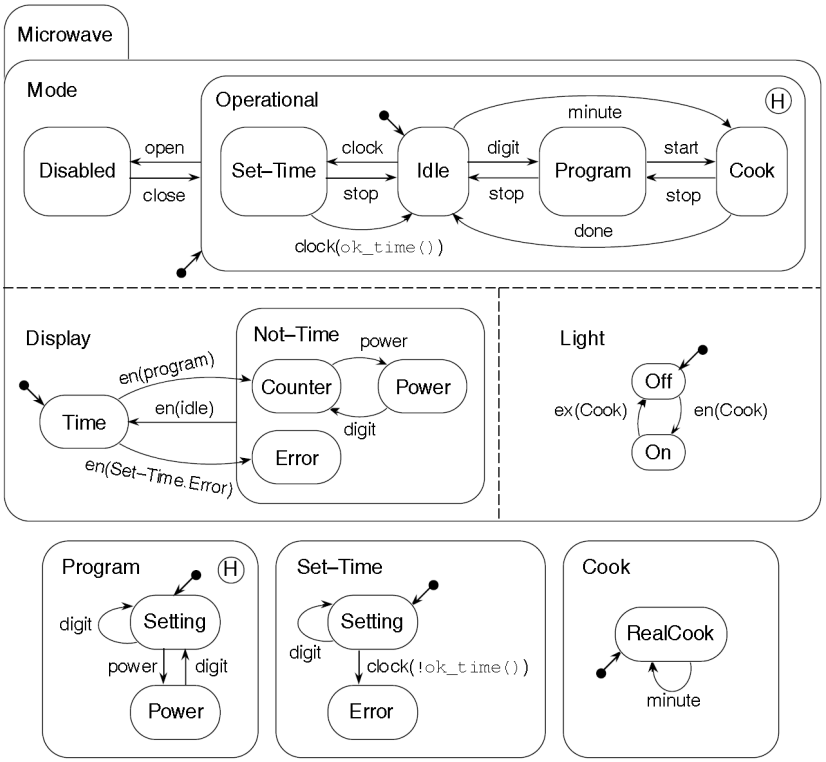

Figure 4.24: Statechart for the Microwave Example

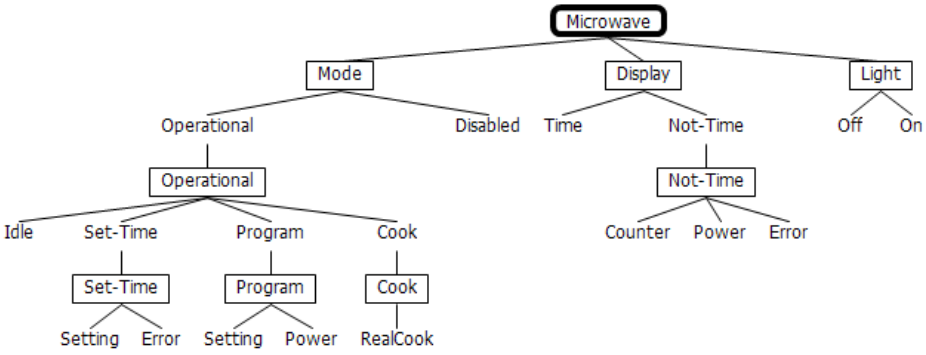

Figure 4.25: Classification Tree for the Microwave Example

For the three orthogonal regions *Mode*, *Display* and *Light*, there are three classifications in the tree. The concurrency of the statechart ideally maps onto those classifications which are also "active" in parallel in test cases. From each region, there can only be one active state which applies for the classes of different classifications as well. The hierarchical states are mapped onto classes and classifications with the same name in the classification tree. The additional classification is necessary, as classes can contain only classifications (and compositions) as refinement but no direct descendant classes.

Although this mapping is quite nice and intuitive, now there are details in the tree missing (Figure 4.26): It does not contain any transitions, no start transitions and no information on (deep) history. As there are no conditions, there are no details on transition guards (conditioned transitions/transitions with dependencies) either.

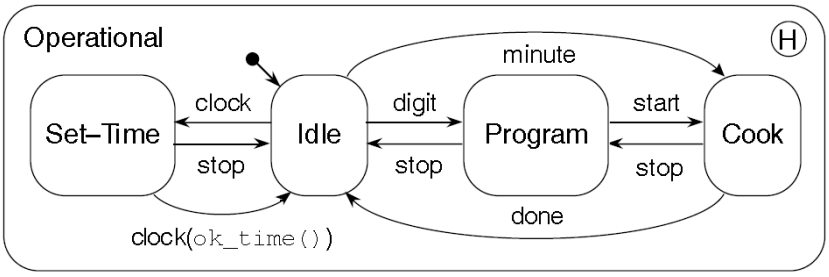

Figure 4.26: State "Operational"

To overcome these problems, we will link these details with the respective tree elements (Figure 4.27).

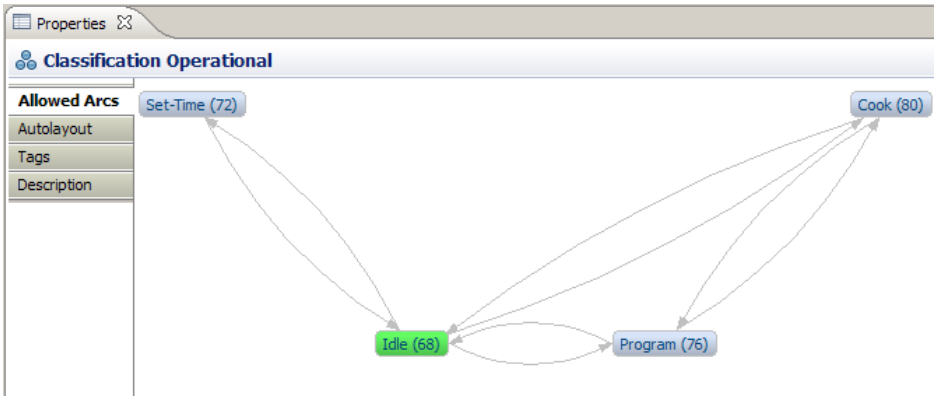

Figure 4.27: Annotated Details for the Classification "Operational"

Example: The classification "Operational" is shown with all allowed transitions and given start node. Constraint handling will be illustrated in the next parts.

The conversion algorithm from statecharts to classification trees is given next (Figure 4.28).

```
 1: create rootNode with Name of statechart
 2: readState (statechart, rootNode)
 3:
 4: readState(state, treeItem)
 5: if state is concurrent then
 6:     for all for each partition do
 7:         create a classification with name of partition
 8:         readChildren (partition, classification)
 9:     end for
10: else
11:     // state is non-concurrent
12:     create classification with name of parent state
13:     readChildren (state, classification)
14: end if
15:
16: readChildren (state, classification)
17: if state has history then
18:     classification set history (true)
19: end if
20: for all child of state do
21:     create one class
22:     readState (child, classification) // recursion
23:     if child is start then
24:         classification add start (child)
25:     end if
26:     for all transition of child  do
27:         classification add transitions (transition)
28:     end for
29: end for
```

Figure 4.28: Statechart to Classification Tree Algorithm

First, the root node of the tree is named after the state machine. The *readState*-method is then called with both the current state, which is the state machine itself for the first call, and the current tree item, in this case the root node. The *readState*-method basically differs only between concurrent and non-concurrent states (Line 5). For concurrent states, it creates a classification for each partition with the name of this partition (Line 7). All children are then added to the corresponding classification. For non-concurrent states, a new classification with the name of the parent state is added, resulting in a classification with the same name as its parent tree item (Line 12). Children of non-concurrent states are then added to this single classification.

The *readChildren*-method first checks whether the state has a (deep) history and stores this information in the corresponding classification. For each child of the state, it creates a class element in the classification tree (Line 21). Each child is then recursively processed by calling the *readState*-method. If one of the children is marked as start state (it has a start transition), it is added to the list of possible start states in the parent classification. All outgoing transitions of each child are also stored in the classification.

In contrast to an earlier approach [HHS03], we can read more details from the formal specification. This way, information on model internals such as constraints and transistions can be preserved when building the classification tree, reducing manual work by the tester. The additional details will then be used for the actual test sequence generation (Section 4.6.5).

4.6.4 Conversion of Classification Trees to Statecharts

For the actual test sequence generation from classification trees, we make some default assumptions:

- In any given plain classification tree, there are no transitions between classes, resulting in an unconnected graph. Test sequence generation will lead to sequences with only one single test step. These test suites are similar to conventional test case generation. Transition coverage is, of course, not available.
- We allow all classes to be reached at start.
- Classifications do not have a (deep) history.

The conversion algorithm from classification trees to statecharts is given next (Figure 4.29).

The *build*-method takes a tree item as the input parameter and returns its corresponding state machine. First, a list of child states is prepared by recursively building all children of the current tree item.

The *build*-method then distinguishes between several cases: If the current tree item is a class or a composition (Line 9), it distinguishes again between the number of children. If a class or a composition has more than one child (Line 10), then this tree item is a parallel state and all children are partitions of it. If a class or a composition has exactly one child (Line 22), it is skipped by directly adding all the child's children to the current tree item. In all other cases, e.g. the tree item is a class or a composition without any children or the tree item is a classification, the prepared list of children is used as the result's list of children (Line 28).

All transitions and possible start states stored in the classification are read and the result is returned.

4.6.5 Algorithm

The algorithm takes the first and second step of our generally defined workflow. The nodes are the classes and classifications from the classification tree. Simple refinement classifications are skipped. Valid transitions can directly be parsed from the annotation in the tree. For the last two steps of the work flow, we propose the following solution.

We use two phases and two kinds of agents to traverse the tree. Travelling is done in such a manner that only valid paths are taken and that all travelled paths together already result in the desired coverage so that there is no need for subset selection.

```
 1: build (treeItem)
 2: state = new State
 3: List children = new List()
 4: for all child of treeItem.getChildren do
 5:     childState = build(child)
 6:     children.add(childState)
 7: end for
 8: Boolean addChildren = !children.isEmpty()
 9: if treeItem is Class || treeItem is Composition then
10:     if treeItem.children.count > 1 then
11:         addChildren = false
12:         result = concurrentState(result)
13:         List subStates = new List()
14:         for all item of children do
15:             if item is SubState then
16:                 result.addSubState(item)
17:             end if
18:         end for
19:     else
20:         if treeItem.children.count == 1 then
21:             addChildren = false
22:             result.setChildren(firstChild.getChildren())
23:             result.setPosition(firstChild.getPosition())
24:         end if
25:     end if
26: end if
27: if addChildren then
28:     result.setChildren(children)
29: end if
30: readArcs(treeItem, result)
31: return  result
```

Figure 4.29: Classification Tree to Statechart Algorithm

For realization, we introduce two kinds of agents: The walker agent and the coverage agent. Both agents will cooperatively traverse the statechart following the algorithm given in the next section.

Walker agents: The task of this agent is to literally walk through the statechart. The walker agents are very simple programs. There is only one kind of walker. Walkers do not have a special order; all of them have the same importance. Walkers can have different lifecycles. They are created and removed on demand. There typically is one walker per active state. Walkers can recognize whether they are stuck, which means that there is no valid transition available. Walkers have a list of child walkers, which can be empty.

Coverage agents: The coverage agents are more sophisticated than the walker agents. Their task is to measure all current and previous coverage levels and to guide the walkers though the statechart. There are different kinds of coverage agents, one for each type generation-rule term introduced in Section 4.6.1: State-coverage, transition-coverage, state-pair-coverage, transition-pair-coverage, and so on. We define an order for all coverage agents by a) their complexity and b) the number of

parameters, which is the number of scopes they cover. The lifecycle of the coverage agents start with the beginning of the generation process. They remain active until the coverage criterion they handle is finished. There is one coverage agent per generation rule term/component, e.g. the rule

$$state\text{-}pairwise(a,b) + transitions(a,b,c)$$

results into two coverage agents.

Coverage agents can determine whether they are finished, both globally and locally. From the example above, we can explain complexity and number of parameters needed for ordering the coverage agents. We use the following formula to calculate the order of coverage agents:

$$i = |order| * |parameters|$$

If two terms have the same index value, we will prefer the term being early in the generation rule formula which can result into missing commutative property for certain generation rules.

Walker algorithm: The basic idea of the algorithm (Figure 4.30) is that all walkers will, one by one, ask the most complex remaining coverage agent where to go next. By walking the route proposed by the most complex coverage agent first, chances are high to cover elements needed by simpler coverage agents, too. For example, transition coverage for a statechart already implies state coverage, too.

The very first step is the creation of coverage agents from the generation rule. Then, the root node (*statechart*) is selected. This means, a walker is created for the root node. For each classification (*parallel section*) under the root node, the class (*node*) with the start transition is selected (Lines 4-9). There will consequently be one walker per classification now. The selection of start nodes is repeated as long as a selected class (*state*) has at least one refinement (*inner states*). A new walker is then created for each refinement step and will be added to the list of child walkers.

As soon as all walkers without child walkers reach an atomic class, the first test step is created. This test step is then checked against the conventional dependency rules. If it is valid, then the test step is added to the test sequence (Line 12) and the coverage measure is updated (Line 13). If it is invalid, test sequence generation is canceled and an empty result set is given (Line 15) as there are no valid test sequences available from this specification.

As long as generating, each walker without child walkers will now perform the following steps. It first identifies (Line 20) the most complex remaining coverage agent. It will then ask this agent where to go next or to stay (Line 21). The candidate class is then checked against conventional and new dependency rules together with all walkers already moved in this turn. If the candidate is valid, we continue with the next walker until all walkers have walked. We then add this test step to the test sequence (Line 32) and update the coverage. We then start the next turn of walking walkers. If the candidate node does not offer any valid test step, the walker will take the second best move from the coverage agent candidates and validates it

again until a valid test step candidate is found. If all candidates fail, the walker will try to stay where it is. If that is not possible, it can still try to step out, which means, its parent walker will move. If even that is not an available option, we start backtracking (Line 35).

```
 1: create coverage agents from rule
 2: select root node // statechart
 3: create walker
 4: for all classification // parallel section do
 5:    repeat
 6:       select (inner) start class // node
 7:       create walker, add to walker child list
 8:    until class is atomic
 9: end for
10: compose test step from selected classes
11: if valid then
12:    add test step to sequence
13:    update coverage
14: else
15:    cancel generation with empty results
16:    finished = true
17: end if
18: while not finished do
19:    for all walker without child-walker do
20:       find coverage agent
21:       ask coverage agent where to go or stay
22:       decide if stuck
23:       if walk transition then
24:          while entering a refined class do
25:             select (inner) start class
26:             create walker, add to walker list
27:          end while
28:       end if
29:    end for
30:    compose test step from selected classes
31:    if valid then
32:       add step to sequence
33:       update coverage
34:    else
35:       backtracking
36:    end if
37: end while
```

Figure 4.30: Test Sequence Generation Algorithm, Phase 1

The candidate from the previous walker is rejected now and the next best choice from this previous walker is taken. This is done recursively until one valid candidate can be found. If there is no candidate—the first walker does not find a valid option—the statechart is globally stuck. It will be reset to its initial state by turning it off and on again.

If adding test steps to the test sequence does not increase coverage for a certain number of steps, we will reset the statechart, as well, and start a new test sequence.

We wait as many steps as there are (inner) classes in the largest classification. All steps without progress are removed from the sequence before adding it to the result test suite.

We repeat this until the global coverage is completed or resetting and starting a new sequence does not increase coverage for a certain number of test sequences; then we can stop. We again wait as many test sequences as there are (inner) classes in the largest classification. In this case, we start a second phase for hard to reach configurations.

Second phase: The algorithm for the second phase for hard to reach configuration is given in Figure 4.31. The approach is inspired by [GFL+96].

```
1:  for all coverage agent (in descending order of importance) do
2:    for all not yet covered item do
3:      find path from item to start in a reverse breadth first search
4:      if there is valid path then
5:        add test sequence of path to result set
6:      else
7:        drop item from coverage measure
8:      end if
9:    end for
10: end for
```

Figure 4.31: Test Sequence Generation Algorithm, Phase 2

The approach is rather simple here. For all coverage agents we get all not yet covered items. We apply a reverse breadth first search for paths from this item to possible start states (Line 3). If a valid path is found, it is added to the result set (Line 5). If there is no valid path, the item is not reachable and is dropped (Line 7).

The approach guarantees the coverage of all coverable items. The result set, however, might not be minimal.

Coverage algorithm: The rating algorithm works as follows (Figure 4.32). It gets a candidate state or transition. For state coverage, self transitions are ignored and zero is returned. Otherwise, it then adds this candidate to a queue together with a weight factor, with an initial weight factor of one. The initial rating is set to zero. The candidate is added to the list of rated items. Then while the queue is not empty the algorithm polls the next state and weight factor from the queue. If the polled node is the original candidate and if the rating is larger than zero, the algorithm has found a loop path with new items. This loop path is preferred by adding the value of 100 to the rating. In this case, or when the current item is on the list of rated items, the while loop goes to its next cycle. Else this node is added to the list of rated items. If the node is on the list of target states (it has not been used in any test step before), the algorithm adds 10 times the weight factor to the result rating. Then, if there are outgoing transitions, child nodes, or subsections, the weight factor is multiplied with a punishment value of 0.95. Target states of outgoing transitions, child nodes and subsections are then added to the queue together with the new weight factor. When the queue is empty the rating is returned.

```
 1: candidate state or transition
 2: if state coverage && self transition then
 3:     return  0
 4: end if
 5: weight = 1.0
 6: rating = 0
 7: queue += (candidate, weight)
 8: while !queue.empty do
 9:     (item, weight) = queue.poll
10:     if item == candidate && rating > 0 then
11:         rating += 100
12:         continue
13:     end if
14:     if ratedItems contains then
15:         continue
16:     end if
17:     ratedItems += item
18:     if targetNodes contains item then
19:         rating += 10 * weight
20:     end if
21:     if item has (outgoing transition || childnodes || subsections) then
22:         weight *= 0.95
23:     end if
24:     for all (outgoing transition && childnodes && subsections) of item do
25:         queue += (item, weight)
26:     end for
27: end while
28: return  rating
```

Figure 4.32: Rating of Candidates

For pairwise coverage, the rating algorithm needs to take into account all walkers at once. When rating moves for state pairwise, the rating algorithm uses two lists, one containing all positions of walkers which will be moved and one list of positions of walkers that have already been moved in this turn. It sends walkers one at a time to states that cover state pairs together with all walkers in the statechart.

For transition pair coverage, the rating algorithm uses a list of all transitions already taken in a move together with the list of positions of walker which still can move during this turn. The rating algorithm then sends walkers through the statechart in such a manner that joint moves of walkers cover transition pairs.

The remaining part of the pair rating works just like the rating of states and transitions.

5 Evaluation

After designing and implementing our approaches, we evaluate them in this chapter. We first apply a prioritized benchmark to both the prioritized pairwise combination and the plain pairwise sorting approach (Section 5.1). We then apply a large set of standard benchmarks to our new deterministic test case generation approach (Section 5.2). We introduce a set of case studies for test sequence generation and evaluate our new test sequence generation approach (Section 5.4).

5.1 Prioritized Generation

Here, we evaluate the impact of weight consideration on weight coverage and on the absolute size of the generated test suites. A first impression is given in Figure 5.1.

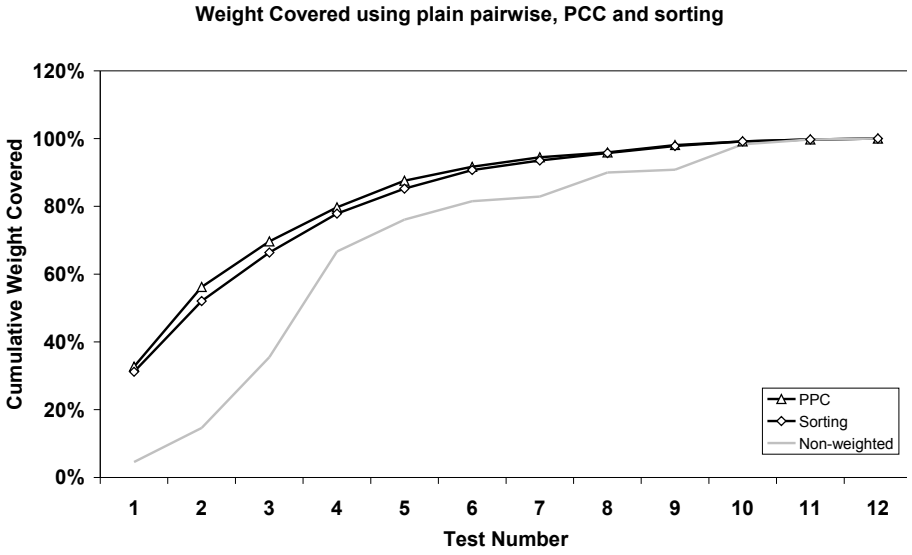

Figure 5.1: Comparison of PPC, Sorting, and of Non-Weighted Generation

The Figure shows a comparison of PPC, Sorting and a non-weighted generation for the the ACC example using the usage model. As can be seen, PPC and sorting perform similar while the unsorted test suite does not perform that well. For a more detailed and systematic evaluation, we use the set of benchmarks proposed in [BC06]: We compare (1) our PPC approach with our sorting approach and (2) PPC with the

deterministic density algorithm (DDA) given in [BC06]. The given benchmark uses four different weight distributions applied to eight scenarios. The distributions are:

- d_1 *(Equal weights)*—All levels have the same weight.
- d_2 *(*50/50 *split)*—Half of the weights for each factor are set to `0.9`, the other half to `0.1`.
- d_3 *(*$(1/v_{max})^2$ *split)*—All weights of levels for a factor are equal to $(1/v_{max})^2$ where v_{max} is the number of levels associated with the factor.
- d_4 *(Random)*—Weights are randomly distributed.

The scenarios $s_1, ..., s_8$ with their resulting test suite sizes are given in Table 5.1. The factors are given in a shorthand notation, for example s_5 with $8^2 7^2 6^2 2^4$ consists of 2 factors with 8 values, 2 factors with 7 values, 2 factors with 6 values, and 4 factors with 2 values. The columns $d_1, ..., d_4$ provide the different results for the PPC runs. The *sort* column reports the resulting test suite size for the sorting approach regardless of the distribution.

Table 5.1: Benchmark Scenarios Result Sizes

	Factors	d_1	d_2	d_3	d_4	sort
s_1	3^4	10	12	10	12	9
s_2	10^{20}	218	228	218	229	245
s_3	3^{100}	130	51	130	52	34
s_4	$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$	92	100	109	106	94
s_5	$8^2 7^2 6^2 2^4$	71	78	86	80	74
s_6	$15^1 10^5 5^1 4^1$	172	190	177	187	177
s_7	$3^{50} 2^{50}$	70	42	84	33	29
s_8	$20^2 10^2 3^{100}$	400	403	420	404	400

5.1.1 Comparison of PPC vs. Sorting

Size. The test suites generated using the PPC are larger than those generated using the sorting approach (Table 5.1). The PPC-generated scenarios s_3 and s_7 in d_1 and d_3 are up to 300% larger than their sorting counterparts. The problem here seems to be the combination of scenario and distribution. For any other distribution or scenario, the PPC results are much smaller and, therefore, closer to the sorting results. For the rest, the resulting PPC test suites are up to 50% larger in some cases (s_3 d_4 or s_7 d_2). On average, the PPC test suites are 42% larger than the sorting test suites. Ignoring the extreme values (s_3 and s_7 from d_1 and d_3), the PPC test suites are 12% larger on average.

Applying equal sorting (d_1), the PPC algorithm results in smaller test suites for some scenarios. For s_2, it is generally smaller. The normal test case generation serving as input for the test case sorting seems to have problems with this particular

scenario. To reveal the reasons behind this behavior, further investigation is needed. For s_6 d_3, the PPC test suite has the same size as the sorting.

For the majority of scenario-distribution combinations, sorting results in smaller test suites, which is not surprising since in prioritization selection focuses on class weights first; in contrast to sorting, which tries to get the smallest test suite available.

Weight coverage. The detailed results are given in Table 5.2. The first column gives the desired weight. The second column gives the scenario. The remaining four columns give the number of test cases needed to reach the weight for each distribution. The values for both algorithms, PPC and sorting, are given next to each other with the smaller value highlighted in bold font.

In d_1, sorting starts very strong. Reaching 25% coverage is always better than or equal to PPC. For 50% and 66%, PPC is only better in s_2 where sorting seems to have a general weakness, and s_8. For the rest of measured weights, PPC becomes better and better, while sorting loses its advantages from the lower values.

In d_2, the prioritization performs better in all scenarios for the 25%–75% target weights. Starting at 90%, sorting gets better with s_1 and s_8. For 95%, sorting also needs fewer test cases with s_4. This case needs further investigation, maybe it is just a good weight distribution for the test case generator creating the PPS input. For 99%, sorting only performs best for s_1.

In d_3, sorting comes close to the PPC. For 25% coverage, both approaches perform equally. For 50%, the PPC is only better for s_2 and s_7. For 66% weight coverage, the prioritization is better for s_2, s_4, s_6, and s_8. For 75% coverage, PPC is always better for large scenarios (s_2, ..., s_7). For 90% and 95% coverage, sorting performs equally both for s_7 and s_3 for the latter. For 99% coverage, sorting surpasses PPC for half of all scenarios.

In d_4, the prioritization approach provides the best weight coverage. Random distribution of weights leads to a high weight coverage when performing test case composition with its pair selection.

In general, the PPC gives better weight coverage. There are, however, three exceptions: Very small scenarios, problematic distributions, and very high weight marks. For *very small scenarios*, sorting has a good starting point since any pairwise-covering test suite has a good chance for containing combinations of any class pair, even combinations of only high-weight pairs. In these cases, sorting can sort the good combinations to the beginning. The influence of *problematic distributions* has already been analyzed in detail earlier. PPC performs better on random and $(1/v_{max})^2$ split while it has some problems on equal distributions (with low target covering marks) and is on par with the sorting approach on the 50/50 split. For higher target marks on equal distributions, it becomes better again since sorting has a general problem here. For *very high weight marks* starting at coverage around 95% or 99% or even higher, PPC loses its advances gradually. Since sorting performs better for 100% coverage, there obviously must be a point $\leq$ 100% where both approaches perform equally.

To conclude. Having two algorithms which both generate test suites covering all

Table 5.2: Benchmark Detailed Result

weight		d_1	d_2	d_3	d_4
25%	s_1	3 3	1 1	3 3	2 2
	s_2	27 27	**9** 12	27 27	**12** 19
	s_3	3 3	**1** 2	3 3	**1** 3
	s_4	11 **10**	**3** 4	4 4	**6** 7
	s_5	8 **7**	**2** 3	2 2	**3** 4
	s_6	23 **22**	**7** 9	12 12	**11** 14
	s_7	2 2	**1** 2	2 2	**1** 2
	s_8	3 3	1 1	3 3	**2** 3
50%	s_1	5 5	**1** 2	5 5	3 3
	s_2	**56** 60	**18** 36	**56** 60	**31** 47
	s_3	6 6	**1** 4	6 6	**3** 5
	s_4	24 **22**	**8** 11	8 8	**15** 18
	s_5	18 **17**	**6** 8	4 4	**8** 10
	s_6	52 **49**	**17** 24	26 26	**27** 36
	s_7	4 4	**1** 3	**3** 4	**2** 4
	s_8	**8** 9	3 3	6 6	**5** 8
66%	s_1	7 **6**	**1** 3	7 **6**	5 5
	s_2	**79** 89	**27** 60	**79** 89	**50** 74
	s_3	8 8	**1** 7	8 8	**5** 8
	s_4	36 **34**	**12** 17	**12** 14	**23** 28
	s_5	28 **27**	**10** 15	7 7	**14** 17
	s_6	74 74	**25** 39	**38** 41	**43** 57
	s_7	6 6	**1** 4	5 5	**4** 6
	s_8	**13** 15	**7** 8	**8** 9	**10** 15
75%	s_1	8 **7**	**3** 4	8 **7**	6 **5**
	s_2	**95** 110	**36** 79	**95** 110	**65** 95
	s_3	**9** 10	**4** 8	**9** 10	**7** 10
	s_4	45 **42**	**16** 23	**16** 19	**29** 36
	s_5	35 **34**	**13** 21	**9** 10	**19** 24
	s_6	**89** 92	**31** 53	**46** 53	**56** 73
	s_7	**7** 8	**2** 5	**6** 7	**5** 7
	s_8	**18** 21	**12** 13	**9** 11	**15** 21
90%	s_1	9 9	7 **5**	9 9	8 **7**
	s_2	**132** 162	**87** 131	**132** 162	**104** 150
	s_3	**15** 16	**9** 13	**15** 16	**12** 15
	s_4	64 **63**	**36** 38	**30** 35	**46** 56
	s_5	**50** 51	**28** 37	**15** 19	**33** 41
	s_6	**123** 131	**73** 92	**83** 92	**90** 111
	s_7	**11** 12	**6** 9	10 10	**9** 11
	s_8	64 **46**	30 30	**15** 19	**35** 43
95%	s_1	10 **9**	8 **7**	10 **9**	10 **8**
	s_2	**152** 190	**121** 163	**152** 190	**129** 181
	s_3	20 20	**12** 16	20 20	**15** 19
	s_4	74 74	53 **51**	**42** 46	**59** 67
	s_5	**57** 59	**43** 46	**23** 25	**44** 50
	s_6	**139** 149	**106** 114	**107** 120	**112** 131
	s_7	**14** 15	**9** 12	13 13	**11** 14
	s_8	92 **62**	56 **47**	19 **26**	**55** 60
99%	s_1	10 **9**	11 **8**	10 **9**	11 **9**
	s_2	**180** 228	**169** 212	**180** 228	**169** 223
	s_3	37 **26**	**19** 23	37 **26**	**21** 26
	s_4	**86** 88	**76** 77	72 **70**	82 82
	s_5	**66** 68	**61** 62	**52** 54	**60** 65
	s_6	**159** 169	**148** 153	**149** 158	**148** 160
	s_7	**20** 21	**14** 18	21 **19**	**17** 21
	s_8	145 **120**	**114** 122	**28** 45	**120** 137

possible pair combinations, the PPC covers weights better than the sorting approach because it tries to combine high weight pairs into early test cases. The sorting approach is worse because it has no influence on the actual composition of test cases with their contained pairs. So while both test suites contain all possible class pairs, PPC results in early weight coverage.

5.1.2 Comparison of PPC with DDA

Size. Comparing PPC and DDA test suite sizes, there is no clear result. From 32 test suites generated with DDA and PPC, 18 DDA suites are smaller than the PPC test suites. For the remaining 14 scenario-distribution combinations, PPC generates smaller test suites. As already stated, PPC produces a very large test suite for the s_3 and s_7 in d_1 and d_3 combinations. The DDA produces smaller test suites for these combinations, similar to the sorting results. These special cases need further investigation. The DDA, however, has two outliers with s_2 and s_6: The result set is 50% larger for s_2 and 25% larger for s_6, compared with PPC.

For the majority of results, both algorithms perform similarly. For d_2, the PPC has some advantages, for d_1 and d_3 DDA performs better. The scenario s_6 seems to be a good PPC scenario, while s_3 and s_7 are handled well by DDA. There is no general tendency for one or the other to produce considerably different test suite sizes since both algorithms aim to cover high weight instead of generating small test suites.

Weight coverage.

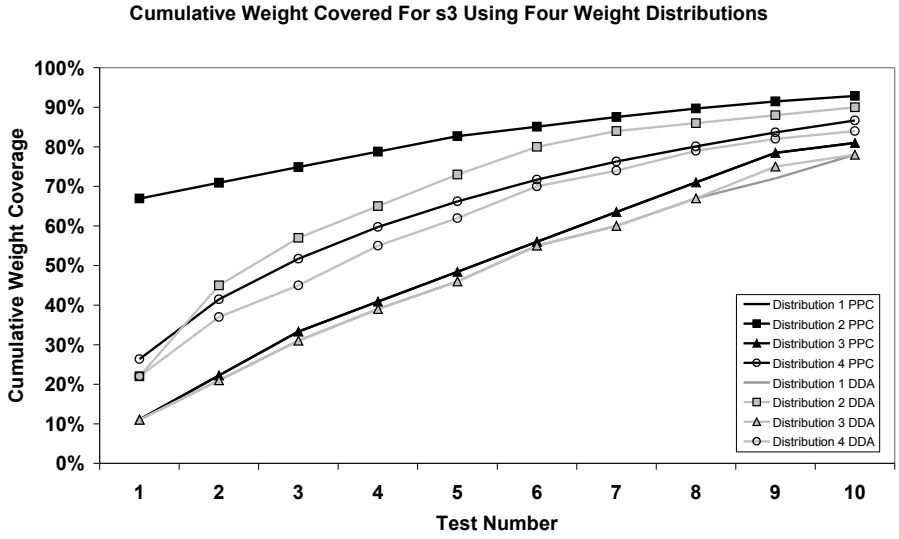

Figure 5.2: Cumulative Weight Covered in the First 10 Tests Using Input s_3

In [BC06], the weight results are only given for s_3, s_4, s_5, s_7, and s_8. We analyzed the given figures and the results are given in Figures 5.2–5.6. The solid black lines

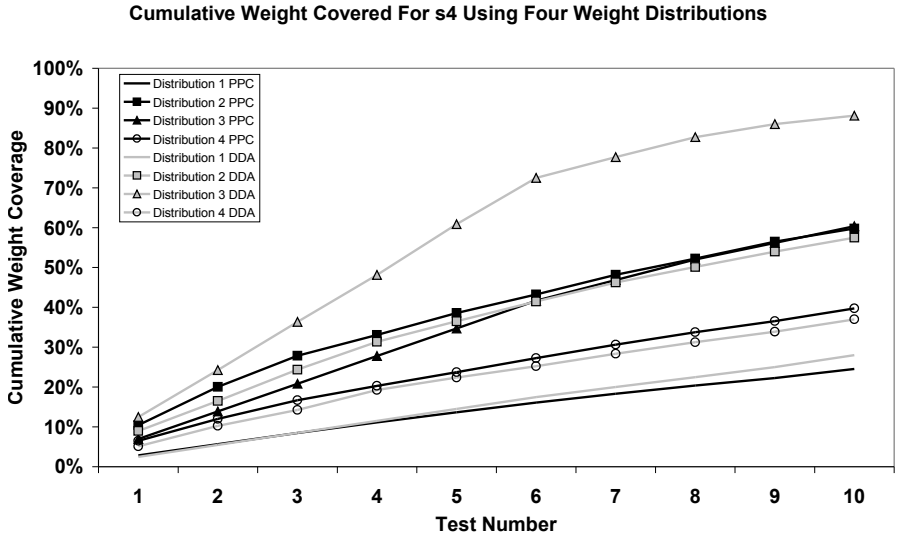

Figure 5.3: Cumulative Weight Covered in the First 10 Tests Using Input s_4

represent the PPC results while the light gray lines represent the DDA results. Both black and gray lines without any markers stand for d_1; d_2 lines carry small solid squares; the solid triangles represent d_3; and d_4 has circles.

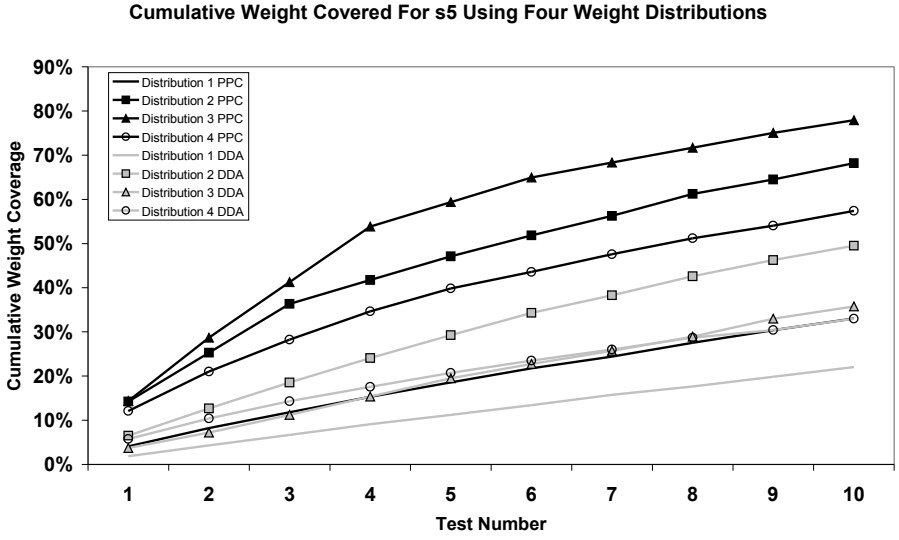

Figure 5.4: Cumulative Weight Covered in the First 10 Tests Using Input s_5

For s_3, PPC works better than DDA for all distributions (Figure 5.2). For d_2, the advantage is remarkably high at the beginning, although at later test cases, DDA

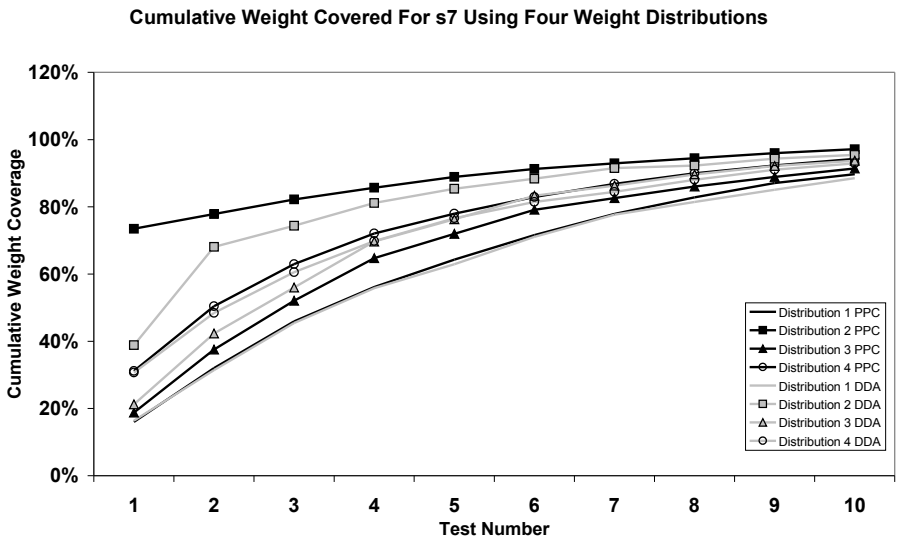

Figure 5.5: Cumulative Weight Covered in the First 10 Tests Using Input s_7

approaches the PPC values.

For s_4, the DDA has a clear advantage for d_3 and a small advantage for d_1 (Figure 5.3). For d_2 and d_4, PPC performs slightly better.

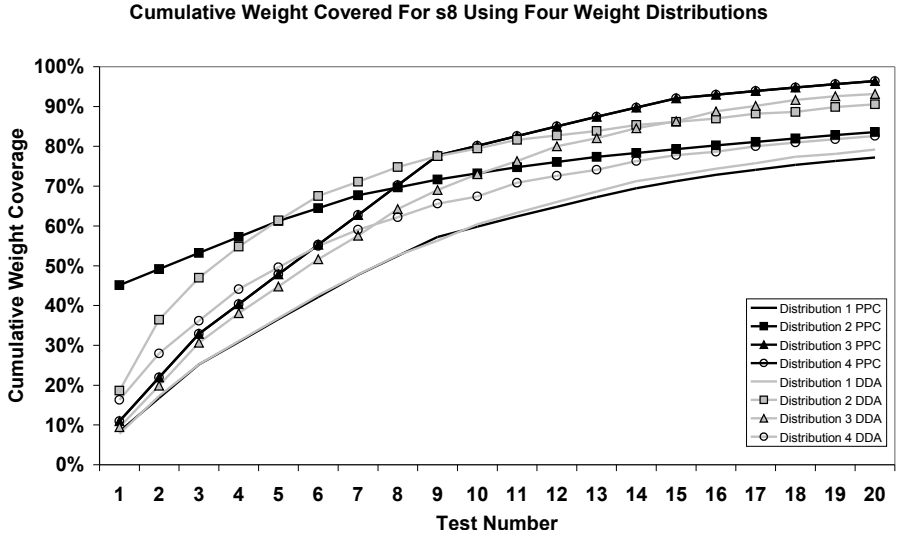

Figure 5.6: Cumulative Weight Covered in the First 20 Tests Using Input s_8

For s_5, the PPC generally performs better than the DDA (Figure 5.4). In this scenario, PPC gives its best results for the d_3.

For s_7 (Figure 5.5), the results are very similar to s_3. The PPC generally performs better than the DDA. The PPC has a very strong start for d_2 while the DDA start is quite similar to the other distribution starts.

For s_8, the DDA starts better than PPC for d_4 (Figure 5.6). For the remaining distributions, the PPC starts better, d_2 again very much better. In later test cases, the DDA surpasses PPC for d_1 and d_2. PPC surpasses DDA for d_4 and stays ahead for d_3 all the time.

Comparing all 240 generated test cases (4 scenarios with 4 distributions with 10 test cases each and 1 scenario with 4 distributions with 20 test cases each) for both DDA and PPC, the PPC performs better for 192 test cases. The DDA performs better for 47 test cases. So while the values for DDA in [BC06] are not 100% accurate, a tendency can be seen: PPC performs better than DDA in 4 of 5 cases with respect to weight coverage. Unfortunately, no results for DDA with s_1, s_2, and s_6 are given. Comparing PPC's good performance in s_6 with DDA results would be quite interesting.

To conclude. PPC performs better than DDA in terms of early weight coverage; the resulting test suites are slightly larger. However, further analysis is needed for the missing scenarios.

5.2 Deterministic Test Case Generation

As already mentioned in Section 4.4 we develop two versions of our algorithm. Version BDD_{POST} checks the tuples that must be covered for validity in phase 2; in version BDD_{PRE} all invalid tuples are excluded in the initialization phase.

For evaluation, we use the benchmarks presented in Table 5.3 that where published in [GCD09]. We use the common notation for benchmarks, i.e., we write n^k if the test problem has k parameters with n values. For constraints, c^m stands for m constraints with c atomic propositions. All experiments are carried out on a PC with an AMD Phenom II X2 555 Processor running at 3.20GHz with 4GB of RAM using Debian Linux 6.0 Squeeze AMD64 (CASA) and Windows 7 64 Bit Edition (ATGT, PICT, our tool). We consider both run time and the size of the result set, and present test set size results in Table 5.4 and run time results in Table 5.5.

Table 5.3: Benchmark Examples

Name	Model	Constraints	Name	Model	Constraints
SPIN-S	$2^{13}4^{5}$	2^{13}	14	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$
SPIN-V	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	15	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$
GCC	$2^{189}3^{10}$	$2^{37}3^{3}$	16	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$
Apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	17	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$
Bugzilla	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	18	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$
1	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	19	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$
2	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	20	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$
3	$2^{27}4^{2}$	$2^{9}3^{1}$	21	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$
4	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	22	$2^{72}3^{4}4^{1}6^{2}$	$2^{20}3^{2}$
5	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	23	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$
6	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	24	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$
7	$2^{29}3^{1}$	$2^{13}3^{2}$	25	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$
8	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	26	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$
9	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	27	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$
10	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	28	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$
11	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	29	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$
12	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	30	$2^{73}3^{3}4^{3}$	$2^{31}3^{4}$
13	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$			

Table 5.4: Evaluation Results, Test Set Size

	related work							our approach	
	ATGT			CASA			PICT	BDD	BDD
	bc	wc	avg	bc	wc	avg		PRE	POST
Spin-S	27	32	29.1	19	23	20.1	26	26	28
Spin-V	44	51	47.6	33	42	38.6	63	45	44
GCC	24	25	24.5	18	28	22.2	30	25	26
Apache	42	44	43	30	37	33.8	40	35	35
Bugzilla	19	23	20.8	16	19	16.7	20	17	17
1	55	62	58.7	37	44	39.7	53	50	49
2	37	42	40.3	30	36	32.6	40	34	34
3	19	22	20.7	18	20	18.2	23	19	19
4	27	31	29.1	20	25	21.9	29	25	23
5	66	72	69	46	66	51.7	65	59	60
6	31	35	33	24	29	24.4	32	29	29
7	11	12	11.7	9	9	9	12	11	11
8	57	65	60.3	38	46	41.4	56	50	50
9	22	25	23.6	20	21	20.1	23	21	22
10	62	66	63.9	42	49	44.4	58	54	55
11	62	66	63.6	40	47	43.2	59	54	55
12	56	60	57.8	37	44	41	56	46	48
13	47	51	48.6	36	43	36.9	44	39	41
14	50	55	52.8	36	40	37.7	48	43	47
15	39	45	42.2	30	39	31.6	39	33	33
16	30	34	31.9	24	29	24.4	29	26	26
17	53	58	55.5	37	46	40.5	54	46	49
18	60	65	61.7	40	47	42.3	58	53	52
19	66	70	67.9	46	70	49.6	66	58	58
20	75	79	77.2	52	56	53.2	70	67	67
21	48	55	51.7	36	39	36.8	48	46	43
22	38	45	42.1	36	37	36.1	39	39	36
23	16	17	16.4	12	15	12.7	18	14	15
24	61	69	64.6	41	49	43.2	57	55	54
25	69	74	71.3	47	57	48.7	65	63	60
26	42	45	44	29	35	31.4	42	39	38
27	44	49	46.4	36	37	36.2	44	38	41
28	70	76	74	49	63	51.6	72	66	63
29	35	38	36.9	29	34	30.9	34	31	31
30	23	28	25.5	16	22	19.9	25	24	22

Table 5.5: Evaluation Results, Execution Times in Seconds [s]

	related work							our approach	
	ATGT			CASA			PICT	BDD	BDD
	best	worse	avg	best	worse	avg		PRE	POST
Spin-S	15.8	17.9	16.6	0.5	37.2	7.8	<1	0.2	0.1
Spin-V	163.5	185.4	177.9	12.3	543.1	72.9	<1	9.2	5.4
GCC	2103.2	2362.7	2249.7	66.1	3409.2	846.7	<1	69.3	20.5
Apache	1533.5	1739.2	1650.1	37.1	194.3	69.3	<1	177.6	100.7
Bugzilla	64.9	71.8	67.6	1.9	15.2	4.4	<1	1.4	0.8
1	433.9	489.2	465.9	25.0	790.3	226.4	<1	43.5	34.8
2	309.7	361.4	337.9	10.5	390.1	31.8	<1	25.6	17.1
3	22.6	25.2	23.4	0.7	5.9	1.2	<1	0.2	0.2
4	100.6	108.3	104.9	3.1	19.0	6.5	<1	2.5	2.2
5	2455.8	2777.1	2627.6	63.6	2963.1	561.0	<1	1565.1	269.3
6	171.1	186.5	178.0	5.8	52.0	15.6	<1	5.2	6.0
7	19.3	20.7	19.8	0.5	1.1	0.6	<1	0.1	0.1
8	667.8	745.7	716.5	40.9	1722.7	316.7	<1	67.3	52.6
9	107.1	114.5	111.0	6.7	23.4	8.9	<1	2.2	6.1
10	1319.1	1538.2	1446.3	56.0	2363.6	524.3	<1	204.0	280.1
11	441.6	506.4	477.5	28.0	1402.7	295.0	<1	32.5	34.9
12	1060.3	1201.5	1147.6	49.1	2278.2	202.2	<1	150.2	116.3
13	714.1	870.9	780.4	22.6	353.1	72.8	<1	77.5	73.3
14	340.4	433.1	359.3	15.1	745.8	81.9	<1	23.2	22.7
15	114.7	164.2	124.4	4.7	70.9	17.4	<1	5.2	4.0
16	229.6	315.9	249.5	7.5	56.9	17.1	<1	7.5	8.6
17	882.7	1043.4	957.4	30.8	3213.2	266.4	<1	169.2	62.7
18	1151.2	1358.1	1280.4	56.7	2199.9	463.0	1	2089.9	199.6
19	3933.3	4545.8	4262.4	89.8	3438.6	592.7	<1	2593.5	572.0
20	2075.3	2374.9	2247.8	236.9	14382.4	2175.3	<1	381.9	286.1
21	273.8	304.5	292.2	13.3	186.4	54.2	<1	14.6	19.4
22	199.6	215.6	209.8	7.4	38.1	11.1	<1	10.1	11.7
23	19.8	21.8	20.8	1.2	8.2	2.7	<1	0.1	0.1
24	671.3	788.4	729.3	39.5	2047.5	447.8	1	67.0	50.8
25	1124.5	1293.8	1223.8	143.5	4386.8	1055.3	<1	254.5	136.6
26	367.9	395.4	382.1	12.4	582.6	75.8	<1	21.1	14.6
27	135.2	157.3	146.7	5.4	34.7	11.2	<1	6.8	5.7
28	3788.3	4315.3	4120.4	157.9	10151.3	1645.5	<1	13198.3	572.3
29	842.9	976.9	919.1	23.5	153.8	48.2	<1	147.8	47.4
30	181.5	205.1	193.5	5.1	124.6	19.9	1	4.4	3.9

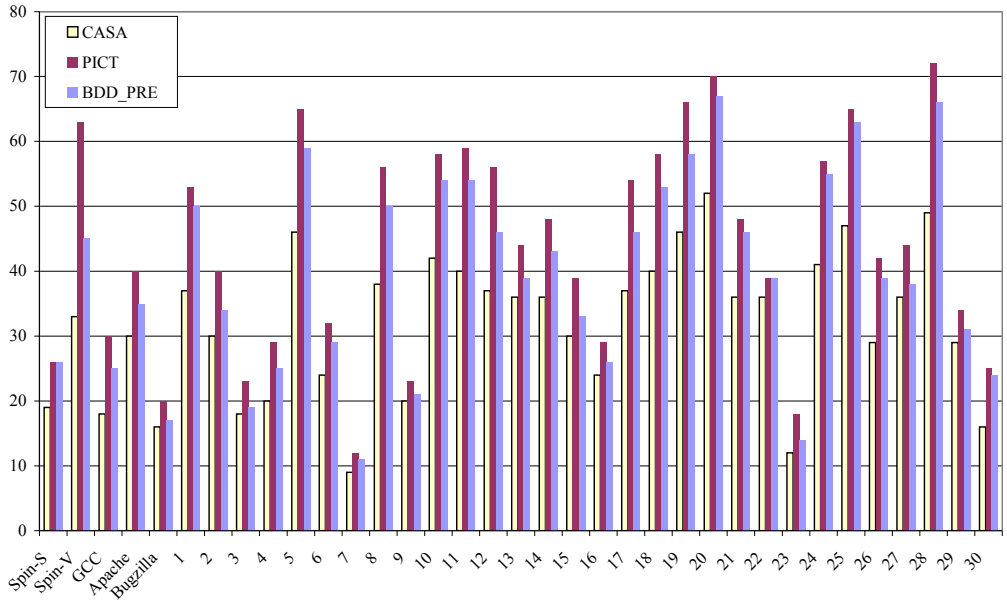

Figure 5.7: Resulting Test Set Sizes for the Deterministic Algorithms and CASA (*CASA* left column—*PICT* center column—BDD_{PRE} right column)

5.2.1 Comparison of BDD_{PRE} and BDD_{POST}

BDD_{PRE} takes slightly longer for nearly all benchmarks. This is due to the fact that we must check more tuples in the initialization phase of the algorithm. The checking process constructs intermediate BDD results that are removed later on by the BDD internal garbage collection. This internal garbage collection causes the increased run time. BDD_{PRE} is faster than BDD_{POST} for only 3 benchmarks. For 27 benchmarks BDD_{POST} terminates faster than BDD_{PRE}.

The evaluation results for both versions of our algorithm do not show obvious differences in terms of test set size. BDD_{PRE} produces smaller result sets for only 11 benchmarks. We expected a greater influence of invalid tuples. We assumed that it would be more difficult to generate good test case candidates with invalid tuples in the second phase of the BDD_{POST} version of our algorithm. In the second phase, invalid tuple identification and candidate generation are critical points that we will investigate further. BDD_{POST} produces smaller result sets for 12 benchmarks, while for the remaining 12 benchmarks both variants produce the same test set size.

5.2.2 Comparison to Other Approaches

We compare our results for pairwise testing with results calculated by ATGT [CG10], CASA [GCD09] and PICT [Cze06]. We choose these tools since, in contrast to other

tools, they offer unrestricted support for constraints. In our experiments we notice considerable variance for the non-deterministic tools. Therefore we present the average case for ATGT and CASA, together with the best and the worst case both for run time and test set size.

We found very large run time differences for CASA for example in benchmarks GCC, 10, 20, and 28. For ATGT, this behaviour was not observed. Examples for remarkable set size differences are benchmarks 1, 21 and 2 for ATGT and benchmarks 20 and 28 for CASA. The most striking results were produced by CASA for benchmark 20. The worst case for run time is about 60 times bigger than the best case, but the difference in result set size is only four test cases (52 to 56). This behavior should be investigated further, since it impacts the tool's applicability.

Our algorithm clearly outperforms PICT in terms of test set size (Table 5.4), but the non-deterministic CASA produces the best results. For 34 out of 35 benchmarks our tool is better than the best case calculated by ATGT. The difference for the remaining one benchmark is not bigger than one test case. PICT is the fastest tool: Calculation times were above one second for only three benchmarks. Our tool is not quite as fast, but it is considerably faster than both non-deterministic tools. The presented results clearly prove the benefit of our approach.

5.3 Test Sequence Generation

5.3.1 Decision Tree Approach

For test sequence generation with the decisuion tree approach, we have not performed a complete set of benchmarks. The approach suffers from state explosion as it needs to unfold all possible parallel states for the generation of internal structures. The apporach can only be used for small examples, but provide valueable insight into the handling of dependency rules and possible aspects of generation rules.

5.3.2 FSM Approach

Only one small example has been evaluated with the result that the approach does not scale well. The example is an Interior Light Control System with four input parameters (Figure 5.8). In the classification tree, there is one classification per parameter. The *Timer* parameter has three possible parameter values; therefore, the *Timer* classification has three attached classes, one per value.

The three remaining parameters *Door*, *Ignition* and *Light*, have two parameter values each, which are *On* and *Off*. There are three conventional dependency rules and seven LTL rules. The conventional rules describe dependencies between different input parameters within a single test step, while the new LTL rules describe constraints between parameters in different test steps.

This small system results in a Büchi automaton FSM with 340 states and 1981 transitions. After interpretation as Test case FSM, 277 states and 1911 transitions

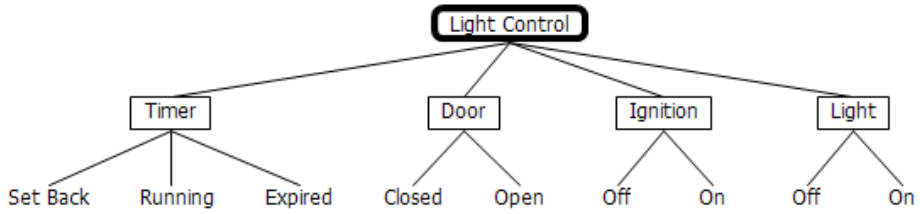

Figure 5.8: Classification Tree for the Interior Light Control System

Table 5.6: Results for FSM based Generation

Length	Size
5	4
6	8
7	16
8	27
9	NA

remain. We had to refrain from the initial idea to offer a visual representation of this internal data structure because of the abundance of state machine elements. For test sequence generation, the maximum length of the sequence has been incremented from five to nine test steps per test sequence (Table 5.6). Results for a sequence length of nine are not available since the algorithm runs out of memory on computation.

The source of state explosion is then further evaluated using a set of simple LTL rules and analyzing their impact on the number of states in the FSMs. It has been analyzed that the FSM is constructed in a way that there is the complete Cartesian product of all possible parameter values ($Timer \times Door \times Ignition \times Light$) per component of the LTL dependency rule. Another problem is the presence of redundant states and transitions in FSMs.

5.3.3 Conclusion

While the approach works, it does not offer the visual insight as it was desired.

The **dependency rules** turn out to work pretty well. They offer a decent granularity to describe all desired scenarios.

The **generation rules**, however, do not offer all details needed. They remain too unspecific and do not reduce the set of valid nodes as desired, resulting in too large test suites. The scalability of both approaches suffers from the explosion of valid nodes, making the approach impractical for large problems.

For practical reasons, we come up with an idea how to reduce the complexity and result size: We add a **filter** on the set of valid nodes by allowing testers to specify specifying the set of desired nodes by taking the result of a conventional test case generation. This enables us to apply a selection criterion on the set of valid nodes

Table 5.7: Results for Statechart Approach

Name	States	Transistions (without start)	Minimal	Pairwise	Complete	State Coverage	Transition Coverage	State Pair Coverage	Trans. Pair Coverage
Keyboard [Mir09]	5	8	2	4	4	2	5	4	16
Microwave [Luc93]	19	23	7	28	56	9	17	29 (2)	43 (5) *(70.1%)*
Autoradio [Hel07]	20	35	11	33	66	13	36	36	44 *(49.2%)*
Citizen [Har87]	62	74	31	108	3121	47	51 *(92.7%)*	87 *(16.2%)*	27 *(2.5%)*
Coffee Machine	21	28	9	27	81	9	18	29	26 *(21.3%)*
Communication	10	12	7	NA	7	7	17	NA	NA
Elevator	13	18	5	20	80	6	9	21 (2)	47 (3) *(96.4%)*
Tetris	11	18	10	NA	10	15	31	NA	NA
Mealy Moore	5	11	5	NA	5	5	24	NA	NA
Fuel Control	5	27	5	25	600	5	12	41 (4)	65 (4) *(75.6%)*
Transmission	7	12	4	12	12	4	9	14 (3)	27 *(50.0%)*
Aircraft	24	20	5	31	625	4 *(86.2%)*	7 (2)	39 (5) *(64.0%)*	22 (6) *(38.6%)*

and it reduces the result set size. The major draw-back of filtering the set of valid nodes, however, is that any "good" pairwise test for example is not sufficient for the generation of test suites with pairwise coverage, since the test case generation does not take the new dependency rules into account.

We therefore propose the statechart approach.

5.4 Statechart Approach for Test Sequence Generation

Details on case studies and results are given in Table 5.7.

The first column lists 12 examples: We use the Keyboard example [Mir09] (as given in Figure 4.23). We found some more examples in literature, a Microwave [Luc93] (Figure 4.24), an Autoradio [Hel07], and of course, Harel's Citizen watch [Har87]. From the IBM Rhapsody examples, we took the Coffee Machine, the Communication example, the Elevator, and the Tetris game. In Matlab Simulink Stateflow, we found Mealy Moore, Fuel Control, Transmission, and Aircraft.

All examples with full details on structure and transitions can be found in Section A.1.

The second and third column list some statistics of the resulting statecharts. Both the number of states and number of transitions are given.

The fourth, fifth and sixth column list the results for conventional test case generation with the classification tree editor for minimal, pairwise and complete combination. Numbers indicate the size of the generated test suite. For examples without multiple classifications, we write *NA* for *not available* in the pairwise column (the Communication, Tetris and Mealy Moore examples), as pairwise test case generation is only available when there are at least two classifications.

The seventh to tenth columns give results from experiments performed for this work. State Coverage and Transition Coverage results are listed in column seven and eight. When there are parallel or hierarchical classifications in the tree, a single test step can cover more than one state. For the keyboard example, the first test step already covers three states, the main *keyboard* state, the *default* state and the *numbers* state. A single number n indicates the size of a single generated test sequence: It is the number of generated test steps n needed for 100% coverage. The second test step in our keyboard example then covers the remaining two states *caps_locked* and *arrows* resulting in a 2 in the State Coverage column. A number n followed by a percentage value (p%) indicates the number of generated test steps n together with a coverage level p% below 100%. In the *Citizen* example for Transition Coverage, 51 test steps cover 92.7% of available transition. When the number n is followed by another number (m), the first number n indicates the total number of test steps while the second number m in parentheses indicates the number of sequences.

In columns nine and ten results for State Pair Coverage and Transition Pair Coverage are given. Again, a single number indicates the number of steps necessary for 100% coverage with a single test sequence. A number n followed by a percentage value (p%) indicates a single test sequence with n test steps reaching p% coverage. A number n followed by another number (m) indicates the number of test steps necessary for 100% coverage using m test sequences. A number n followed by both another number (m) and a percentage value (p%) indicates the number of test steps n distributed over m test sequences resulting in a total coverage level of p%.

The set of case studies is still too small to make final statements on performance with regard to scalability, execution times, and result set sizes. Preliminary results are, however, already very promising.

Execution times: We are not giving detailed figures for generation times here since they are all below 1sec on an Intel Core2Duo with 2Ghz and 3GB RAM running our Java implementation on a single core under Windows XP.

Coverage: Results from the experiments clearly show that our approach is capable of generating test sequences with given coverage levels. For State Coverage, 100% coverage was reached for 11 of 12 scenarios. The remaining scenario only resulted in 86.2% coverage (scenario Aircraft). In all 12 cases, coverage was reached in a single test sequence.

For Transition Coverage, again for 11 of 12 cases 100% coverage has been reached. For the Citizen scenario, only 92.7% coverage was achieved. In 11 of 12 cases, the result only consists of one test sequence, while for the aircraft scenario two sequences were generated. The algorithm had to reset the walker agents to their initial positions to reach missing states.

For State Pair Coverage, 7 of 9 scenarios reached 100% coverage. In the Aircraft Scenario, full Pair Coverage was not reached due to unreachable states in state coverage. The Citizen showed very poor performance with only 16.2% coverage. Further analysis and adaptation is necessary to complete this scenario.

For Transition Pair Coverage, there is only one scenario with 100% coverage which

is the simple Keyboard example. The remaining 8 scenarios show average performance with the Citizen example again being the most difficult one.

Note: Results given here only contain implementation of walker phase one, since phase two was not available yet.

Result set size: For state coverage, only one or two additional steps are needed for 10 of 12 cases in comparison with the conventional minimal combination. Only the Citizen and the Tetris example need considerably more steps to reach state coverage. Not every state can be reached from every other state here but some require additional traversal of other states and, therefore, additional test steps. For the resulting test suite of the Citizen example, there seems to be some potential for optimization.

For state pair coverage, result set sizes are similar to conventional pairwise test generation in 6 of 9 cases. In the Fuel Control scenario, several additional steps are necessary to complete the test suite resulting in 41 test steps for state pairwise in contrast to 25 test steps for conventional pairwise. The remaining 2 scenarios did not complete.

Result sizes of transition coverage are not evaluated in our case studies since there is no comparison available with other approaches. For the goal of test suite minimization we will need to evaluate minimal result suite size using a brute force approach, allowing us to compare our results. For the brute force approach, we expect to see long execution times making it impractical for productive use.

Scalability: We have not yet tested our approach for very large scale case studies. If performance decrease is too large, we might reduce the search depth of coverage agents. The Citizen example already shows scalability issues with state pair and transistion pair coverage.

Extendability: We have successfully implemented test sequence generation for state, transition, state pair, and transition pair coverage. The implementation of pairwise coverage problems is known to be NP-complete [WP01].

Parameter tuning: We have done experiments on the influence of the punishment factor for both state and transition coverage. In a set of 100 experiments each, we have tested all factors from 0.01 to 1.00 for all 12 scenarios. Lower values like 0.1 turns out to be much better than 0.95 as it was selected earlier. However, for one scenario, the Communication example, higher punishment values result in smaller test suites and better coverage. We need to investigate the influence of the other parameters as well.

6 Conclusion

With software as a central part of our everyday life, software testing is the essential part of quality assurance in software development. The testing of conformance with specifications increases the confidence in software quality. However, the complete test of a software system is rarely possible due to the large amount of possible test cases. A good test case design therefore tries to select valuable subsets of this large test case amount, while both avoiding redundancies and combining as many different parameters as possible. The classification tree method allows for the systematic specification of a system under test and its corresponding test cases. It is a common approach based on the systems specification. The classification tree editor implements the combinatorial test design and allows testers to automatically generate test suites for given coverage levels.

Our initial analysis has shown that there was no support for test case prioritization with the classification tree method. We therefore defined values of importance for different test aspects. These values of importance, called weights, can now be used for prioritized test case generation. Resulting test suites contain test cases ordered by their importance, e.g. occurrence probability. Coverage criteria help to optimize the test suite and to identify the quality of the remaining subset. This allows testers to select only the most important test cases and therefore reduce the test case size.

The classification tree method now allows the prioritization of certain test aspects and also offers prioritized test case generation. Resulting test suites can be optimized further to reduce test effort. The test case generation performs better than all other known approaches for test case prioritization in combinatorial testing.

The generation of test cases originally was not deterministic, resulting in a lack of replicability. The original test case generation was based on a random process, and therefore the results from different generation runs varied. Additionally, the original test case generation performed badly with complex dependency rules or compositions of contradictory rules. A deterministic test case generation was therefore desired to produce exactly the same result for the same generation problems and to handle dependency rules neatly.

The new test generation algorithms now processes complex systems with constraints and their results can be reproduced. The performance in terms of both efficiency and effectiveness is better than previous approaches and equivalent to or better than other approaches for combinatorial testing.

To reflect the continuous usage of software, test cases must reflect elements in a certain order. The outcome of one test case is used as the input for the next test case. Test sequences can be used to model these consecutive events. Since it is rarely

possible to test all possible test sequences for practical reasons, test sequence design is a crucial task. In addition to countless configurations for conventional test case design, there are potentially endless different orders and repetitions of test steps in a test sequence. Furthermore, test steps cannot be composed in any arbitrary order as it is required for some configurations of the software that other things have been done first. We defined a way of specifying constraints between test steps of a test sequence and introduced measures of coverage to design automatic generation of test sequences available with the classification tree method. We use a multi-agent approach to implement the actual test sequence generation.

Test sequences can now be generated automatically. We have developed dependency rules to describe constraints between test steps and automated test sequence generation. Continuous combinatorial testing is therefore now available. Our approach, based on a statechart representation of the system under test and using a multi-agent system to travel the statechart, already produces good results.

6.1 Prioritized Generation

We have developed and implemented the prioritized test case generation using qualified classification trees. To allow the prioritization of test aspects, we developed prioritization models and integrated them into the classification tree. For qualification, we introduced three different usage models allowing us to specify the value of importance directly in the classification tree. These weights can be assigned to classification tree elements and are then considered during prioritized test case generation. We extended existing combination rules for test case generation using these priority values and we developed new combination rules taking these values into account. To ease the generation process, we established a mapping of the classification tres and dependency rules onto a logical expression, representing the set of valid test cases in a Binary Decision Diagram. We then use this internal representation for the actual prioritzed test case generation. The prioritization checks this representation to determine whether test cases are valid, and if there can be valid test cases for any combinations of tuples of tree elements. This calculation is available at low runtime cost. Resulting test cases now have a defined importance for the test suite resulting from the priorities assigned to elements of the classification tree used for this test case. Additionally, the introduction of coverage measures allows measuring the coverage level of generated test cases. All generation rules still support dependency rules specified for the system under test in the classification tree. This allow testers to generate test suites with an order of importance specified by the priority values assigned to classification tree elements and to select subsets of test suites that have a defined quality, calculated using coverage rates with the new coverage measures. The prioritized test case generation is then compared with existing approaches. In most cases, our prioritized test case generation performs better than or as well as existing approaches.

A set of benchmarks using eight scenarios with four different weight distributions

has successfully been applied to both algorithms, the prioritized pairwise combination and the sorting approach, demonstrating their usability.

Based on the results, a guideline on when to use which technique can be given: If a full pairwise coverage is already established, sorting can help to select subsets of test suites. If the weight distribution is equal or scenarios are small, there is no reason to use prioritized test case generation. If, however, scenarios are large and distributions tend to be non-uniform, the application of prioritized test generation becomes worthwhile in all cases where subsets of test suites are needed. Then, a subset selection based on weights is more successful using prioritized test generation.

For future work, we see the extension to higher n-wise coverage and a larger set of benchmarks. Future work will also analyze the test generation times for all approaches (PPC, sorting, ...) given in this work.

Having prioritized generation, we use the representation of valid test cases for a new test case generation for t-wise coverage.

6.2 Deterministic Test Case Generation

We developed a new approach for dependency rule handling and a deterministic approach for test case generation. We achieved this by developing a unified representation for both the dependency rules and the classification tree, in order to ease the handling of constraints. This unified representation, a Binary Decision Diagrams, is then used to perform deterministic test case generation. Handling of dependency rules has been improved over existing approaches for test case generation in the classification method. We have then compared our new approach using a large set of benchmarks and compare our result with both previous test case generation of the classification tree method as well as existing approaches for combinatorial testing. Our approach performs better than previous test case generation with the classification tree method in terms of both result set size and generation time. The performance in comparison with other combinatorial test case generation techniques, though, is not always better in terms of result set size and generation time. For specific tasks, e.g. finding the smallest test suite generated using a deterministic approach, our new test case generation is still very good.

In this work we have presented a new approach for combinatorial interaction testing with constraints. The approach is based on our observation that a combinatorial interaction testing problem with constraints can be represented as a single Boolean formula. Valid test cases of the CIT problem and satisfying assignments of the corresponding formula are in a one-to-one relation. We use this relationship in our deterministic test set generation algorithm by selecting satisfying assignments of the formula as test cases. This idea allows for a very elegant integration of arbitrary user-defined explicit constraints and resulting implicit constraints. Binary Decision Diagrams as an efficient representation of Boolean operators guarantee good run time results of our algorithm.

The results of our algorithm on benchmark test problems presented in this paper

prove its practical applicability. Our algorithm outperforms PICT, the only known deterministic algorithm supporting constraints, with respect to result set size. Run time results of our algorithm are adequate for employment with realistic test problems. The new deterministic test case generation performs better than the previous random-based test case generation of the previous classification tree editor as well as the PICT algorithm, the only known deterministic test case generation algorithm supporting constraints.

We are convinced that there is still potential for optimization in our approach. We plan to improve the checking strategy for invalid tuples, since this is the most time-consuming step in our algorithm. Furthermore, we will tune the candidate construction used in the second phase of the algorithm which will decrease the result set size. We will investigate the benefit of integrating a random component into our approach even if we have to renounce determinism. We plan to conduct further case studies to gain deeper insights on the influence of the constraints to further improve our approach.

6.3 Test Sequence Generation

For testing continuous behavior of a system with the classification tree method, we developed the generation of test sequences. There was not much previous work here, we were looking for something like “parallel chinese postman” / “orthogonal chinese postman” or similar techniques.

We have introduced advanced dependency rules to model continuous behavior of the system under test. These advanced dependency rules allow the tester to specify the legal transitions of different states of the system and, therefore, to specify constraints between different steps of a test sequence. Then the test sequence generation produces test suites of test sequences that fulfill the advanced generation rules specifying desired coverage level. For the new test sequence generation with the classification tree method we have developed new advanced test generation rules. We have identified new coverage levels specific to the continuous nature of systems under test to model the advanced generation rules. We have implemented several prototypes using different representation techniques and gained the best results by interpreting classification trees as statecharts with some simplifications. There were no existing benchmark suites available for continuous testing in combinatorial testing so we had to select a suite of benchmarks on our own. In the identified set of benchmark scenarios, test sequence generation performed well in all scenarios we tested.

For the test sequence generation, we identify advanced dependency rules and new generation rules. We again use the representation of valid test cases during the generation process.

We have successfully implemented test sequence generation for the classification tree method using a multi-agent system. The distinction between simple walker agents and sophisticated coverage agents enables us to generate test suites with

desired coverage levels. The set of case studies is still too small to make reasonable statements on performance, regarding scalability, execution times and result set sizes; preliminary results are, however, already very promising. The dependency rules and generation rules turn out to work pretty well. They offer a decent granularity to describe all desired scenarios. Since we split actual traversal of the test problem from the rating of possible paths, we can now easily swap in single parts, e.g. use different guidance in the coverage agents.

Existing approaches that map formal specifications (e.g. Z language [HHS03]) onto classification trees suffer from the loss of structural details such as hierarchies, concurrency or allowed transition, whereas our approach does not.

For future work, we see the extension to higher n-wise coverage, completion of incomplete scenarios and a large-scale set of benchmarks. As already proposed in [KL11], we will use search-based techniques as well to evaluate effectiveness and efficiency of our approach. We will evaluate the influence of parameter tuning for result set sizes and we will consider the limitation of sequence length and favoring of reset or travel back to start, including the cost of a reset.

There are still some missing features in the test sequence generation: The classification tree editor allows specifying properties of transitions, e.g. transition types (linear, spline, sine ...) and timing characteristics. We therefore propose to enhance dependency rules to represent additional transition details. It would then be desirable to automatically generate advanced transitions during test sequence generation also conforming to time and type constraints.

It would be interesting to see how test sequence generation can also take priority values into account.

6.4 Future Work

In addition to several minor adjustments and enhancements to three individual test case generation approaches, we prose two major aspects of future work for all three of them: One is the use of search based techniques for test case generation, the other is the parallelization of all tree generation algorithms.

We propose to investigate the use of multi-objective algorithms in order to combine a test case generation technique, the Classification Tree Method, with a test case selection and prioritization method [KL11]. Yoo and Harman [YH10] have already shown that multi-objective algorithms can be applied to test case selection and prioritization problems. Search-based techniques can also be easily parallelized as CASA already shows [GCD09].

This brings us to the other aspect we like to investigate further: the parallelization of test case generation. Since multicore computers are common nowadays, all three of our approaches are wasting execution time as they work in a single thread only. There is already initial work on multicore BDD by Intel [He09], but it requires reimplementation or at least modification of existing BDD libraries.

We think, however, that reducing the generation times further is crucial for even better acceptance of the generation algorithms and tools implementing them.

List of Figures

List of Tables

Bibliography

[Aml00] Ståle Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287–295, 2000.

[BC06] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.

[BC07] Renée C. Bryce and Charles J. Colbourn. The density algorithm for pairwise interaction testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):159–182, 2007.

[BCM01] Simon Burton, John Clark, and John McDermid. Automated generation of tests from statechart specifications. In *Proceedings of Formal Approaches to Testing Software (FATES) 2001*, pages 31–46, August 2001.

[BE10] Dragan Bošnački and Stefan Edelkamp. Model checking software: on some new waves and some evergreens. *Int. J. Softw. Tools Technol. Transf.*, 12:89–95, May 2010.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BLLP04] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Softw. Pract. Exper.*, 34:915–948, August 2004.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput*, 35:677–691, 1986.

[CC04] Charles J. Colbourn and Myra B. Cohen. A deterministic density algorithm for pairwise interaction coverage. In *Proc. of the IASTED Intl. Conference on Software Engineering*, pages 242–252, 2004.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[CDFY99] Mirko Conrad, Heiko Dörr, Ines Fey, and Andy Yap. Model-based Generation and Structured Representation of Test Scenarios. In *Proceedings of the Workshop on Software-Embedded Systems Testing, Gaithersburg, Maryland, USA*, 1999.

[CDS07] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007.

[CG10] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, April 2010.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[Con05] Mirko Conrad. Systematic testing of embedded automotive software-the classification-tree method for embedded systems (CTM/ES). *Perspectives of Model-Based Testing*, 2005.

[CSR06] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes*, 31:1–9, November 2006.

[Cze06] Jacek Czerwonka. Pairwise testing in real world, practical extensions to test case generators. In *Proceedings of 24th Pacific Northwest Software Quality Conference*, pages 419–430. Citeseer, 2006.

[EFW01] Ibrahim K. El-Far and James A. Whittaker. Model-Based Software Testing. *Encyclopedia of Software Engineering*, 2001.

[EMR02] Sebastian Elbaum, Alexey. G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[ERKM04] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12:185–210, 2004.

[FKCA12] Javier Ferrer, Peter M. Kruse, J. Francisco Chicano, and Enrique Alba. Evolutionary algorithm for prioritized pairwise test data generation. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2012*, Philadelphia, USA, 2012.

[GCD09] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. *Search Based Software Engineering, International Symposium on*, 0:13–22, 2009.

[GFL+96] Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtenstein, Shmuel Ur, and Yaron Wolfsthal. Coverage-directed test generation using symbolic techniques. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 143–158, London, UK, 1996.

[GG93] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[GPV+95] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[GQW+09] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction coverage meets path coverage by smt constraint solving. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, pages 97–112, 2009.

[GR01] Angelo Gargantini and Elvinia Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7:1050–1067, 2001.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[He09] Yuxiong He. Multicore-enabling a binary decision diagram algorithm, 2009.

[Hel07] Steffen Helke. *Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion*. PhD thesis, Technische Universität Berlin, 2007.

[HHS03] Robert M. Hierons, Mark Harman, and Harbhajan Singh. Automatically generating information from a Z specification to support the Classification Tree Method. In *3rd International Conference of B and Z Users, LNCS volume 2651*, pages 388–407, June 2003.

[HRSR09] Andreas Hoffmann, Axel Rennoch, Ina Schieferdecker, and Nicole Radziwill. A generic approach for modeling test case priorities with applications for test development and execution. In *Modellbasiertes Testen (MOTES)–Von der Forschung in die Praxis (2009), 4. Workshop im Rahmen der 39. Jahrestagung der GI*, 2009.

[HRV+03] Mats P.E. Heimdahl, S. Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *Proceedings of Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[Jan05] Ben Jann. *Einführung in die Statistik*. Oldenbourg, 2005.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[KKL10] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. Practical combinatorial testing. Technical report, National Institute for Standards and Technology (NIST), October 2010.

[KL10a] Peter M. Kruse and Magdalena Luniak. Automated test case generation using classification trees. In *Proceedings of StarEast 2010*, 2010.

[KL10b] Peter M. Kruse and Magdalena Luniak. Automated test case generation using classification trees. *Software Quality Professional*, 13(1):4–12, 2010.

[KL11] Peter. M. Kruse and Kiran Lakhotia. Multi objective algorithms for automated generation of combinatorial test cases with the classification tree method. In *Symposium On Search Based Software Engineering (SSBSE 2011)*, 2011.

[KLK08] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, 2008.

[KM05] Alexander Krupp and Wolfgang Müller. Modelchecking von Klassifikationsbaum-Testsequenzen. In *GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, April 2005.

[Kru11] Peter M. Kruse. Test sequence generation from classification trees using multi-agent systems. In *Proceedings of 9th European Workshop on Multi-agent Systems (EUMAS 2011)*, 2011.

[KS12] Peter M. Kruse and Ina Schieferdecker. Comparison of Approaches to Prioritized Test Generation for Combinatorial Interaction Testing. In *Federated Conference on Computer Science and Information Systems (FedCSIS) 2012*, Wroclaw, Poland, 2012.

[KW11a] Peter M. Kruse and Joachim Wegener. Sequenzgenerierung aus Klassifikationsbäumen. In *Proceedings zum 31. Treffen der Fachgruppe TAV der Gesellschaft für Informatik*, Paderborn, Germany, 2011.

[KW11b] Peter M. Kruse and Joachim Wegener. Test sequence generation from classification trees. In *Sistemas e Tecnologias de Informação, Actas da 6ª Conferência Ibérica de Sistemas e Tecnologias de Informação*, Chaves, Portugal, 2011.

[KW12] Peter M. Kruse and Joachim Wegener. Test sequence generation from classification trees. In *Proceedings of ICST 2012 Workshops (ICSTW 2012)*, Montreal, Canada, 2012.

[Lee59] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[LT98] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, HASE '98, pages 254–261, Washington, DC, USA, 1998.

[Luc93] Paul J. Lucas. *An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1993.

[Lun09] Magdalena Luniak. Priorisierende Kombinationsregeln in der Klassifikationsbaum-Methode. Master's thesis, Technische Universität Berlin, 2009.

[LW00] Eckard Lehmann and Joachim Wegener. Test case design by means of the CTE XL. In *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark*. Citeseer, 2000.

[Mir09] Mirosamek. Two orthogonal regions (main keypad and numeric keypad) of a computer keyboard. http://en.wikipedia.org/wiki/File:UML_state_machine_Fig4.png, 2009.

[MN05] Soumen Maity and Amiya Nayak. Improved test generation algorithms for pair-wise testing. In *ISSRE*, pages 235–244. IEEE Computer Society, 2005.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of Ajax web applications. In *In Proceedings of the International Conference on Software Testing (ICST 2008)*, April 2008.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[NL11] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[Obj10] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure Verion 2.3*, 2010.

[OHY11] Jungsup Oh, Mark Harman, and Shin Yoo. Transition coverage testing for Simulink/Stateflow models using messy genetic algorithms. In *Genetic Algorithms and Evolutionary Computation COnference (GECCO 2011)*, pages 1851–1858, July 2011.

[Pun04] Abraham P. Punnen. The traveling salesman problem: Applications, formulations and variations. In *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*, pages 1–28. Springer, 2004.

[Rei10] Robert Reicherdt. Testfallgenerierung mit Binary Decision Diagrams für Klassifikationsbäume mit Abhängigkeiten. Master's thesis, Technische Universität Berlin, 2010.

[Sch13] Henno Schooljahn. Test Sequence Validation and Generation using Classification Trees. Master's thesis, Delft University of Technology, 2013.

[Sed03] Robert Sedgewick. *Algorithms in Java - Part 5: Graph Algorithms*. Addison-Wesley, Boston, MA, USA, 3 edition, 2003.

[SRG11] Elke Salecker, Robert Reicherdt, and Sabine Glesner. Calculating prioritized interaction test sets with constraints using binary decision diagrams. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 278–285, Washington, DC, USA, 2011. IEEE Computer Society.

[Ura92] Hasan Ural. Formal methods for test sequence generation. *Comput. Commun.*, 15:311–325, June 1992.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 1–22, London, UK, 2001.

[Wal11] Nick Walther. Testsequenz-Generierung und Repräsentation mit der Klassifikationsbaum-Methode. Master's thesis, Humbold Universität Berlin, 2011.

[WG93] Joachim Wegener and Matthias Grochtmann. Werkzeugunterstützte Testfallermittlung für den funktionalen Test mit dem Klassifikationsbaum-Editor CTE. In *Proceedings of the GI-Symposium Softwaretechnik '93*, pages 95–102, Dortmund, 1993.

[Wil02] Alan W. Williams. *Software Component Interaction Testing: Coverage Measurement and Generation of Configurations*. PhD thesis, School of Information Technology and Engineering, University of Ottawa, 2002.

[Win08] Andreas Windisch. Search-based testing of complex simulink models containing stateflow diagrams. In *Proceedings of 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008*, pages 251–251, Lillehammer, Norway, 9-11 April 2008.

[Win10] Andreas Windisch. Search-based test data generation from stateflow statecharts. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1349–1356, New York, NY, USA, 2010.

[WLPS00] Guido Wimmel, Heiko Loetzbeyer, Alexander Pretschner, and Oscar Slotosch. Specification based test sequence generation with propositional logic, 2000.

[WP01] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications*, volume 30, pages 301–311, 2001.

[WPT95] Gwendolyn H. Walton, Jesse H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage model. *Softw. Pract. Exper.*, 25(1):97–108, 1995.

[YH10] Shin Yoo and Mark Harman. Using hybrid algorithm for pareto effcient multi-objective test suite minimisation. *Journal of Systems Software*, 83(4):689–701, April 2010.

A Appendix

A.1 Test Sequence Generation Examples

In this section, we provide all examples for the test sequence generation benchmarks.

Each example consists of a classification tree with added ID values for all tree elements to avoid any ambiguity. In square brackets with a leading S : we provide a list of corresponding start nodes for each classification. The list of all transitions from within this classification concerning all contained classes is provided as comma separated list.

Each transition $(a \rightarrow b)$ represents a directed transition from a to b. ID values for all classes are provided for clarity.

Multiple transitions from a to b are provided as often as they occur leading to repetition in the list.

Transitions from a in one classification A to any b in a different classification B are only listed in the classification of origin A.

keyboard.cte

Example Keyboard States [Mir09]: A typical PC desktop keyboard features at least two regions: The *main area* with all characters and a *number pad*. Both regions have their own modes. While the main area has the *caps lock* feature, to input only uppercase letters, the number pad can be switched between *numbers* and *arrows* using the num lock button. Both regions and their states are independent of each other. In the classification tree *keyboard* (Figure A.1), there is one classification per keyboard region. Each classification has two classes representing the corresponding states.

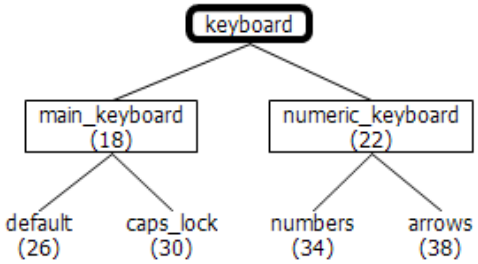

Figure A.1: Keyboard Example

main_keyboard(18)	$[S : default(26)]$:
	$(default(26) \rightarrow default(26)), (default(26) \rightarrow caps\_lock(30)),$
	$(caps\_lock(30) \rightarrow caps\_lock(30)), (caps\_lock(30) \rightarrow default(26))$
numeric_keyboard(22)	$[S : numbers(34)]$:
	$(numbers(34) \rightarrow numbers(34)), (numbers(34) \rightarrow arrows(38)),$
	$(arrows(38) \rightarrow arrows(38)), (arrows(38) \rightarrow numbers(34))$

microwave.cte

Microwave Example [Luc93]: The example microwave device has three independent function blocks. It features a *Light*, *Display* and a *Mode*. All three features are independent of each other. The Light can be *on* and *off* while the Display either shows *Time* or *Not-Time*. If it shows Not-Time, then it shows one of *Counter*, *Power* or *Error*. The Mode can be *Operational* or *Disabled*. If Mode is Operational, then it is one of *Idle*, *Set-Time*, *Program*, or *Cook*.

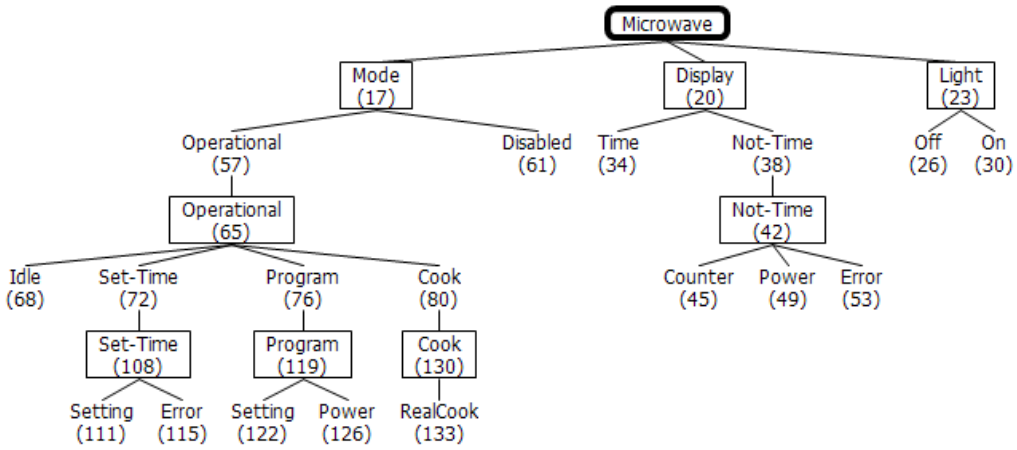

Figure A.2: Microwave Example

Mode(17)	$[S : Operational(57)]$:
	$(Operational(57) \rightarrow Disabled(61)), (Disabled(61) \rightarrow Operational(57))$
Display(20)	$[S : Time(34)]$:
	$(Time(34) \rightarrow Counter(45)), (Not-Time(38) \rightarrow Time(34)),$
	$(Time(34) \rightarrow Error(53))$
Light(23)	$[S : Off(26)]$:
	$(Off(26) \rightarrow On(30)), (On(30) \rightarrow Off(26))$

$Operational(65)$	$[S : Idle(68)]$:
	$(Idle(68) \rightarrow Set-Time(72)), (Set-Time(72) \rightarrow Idle(68))$,
	$(Idle(68) \rightarrow Program(76)), (Program(76) \rightarrow Idle(68))$,
	$(Program(76) \rightarrow Cook(80)), (Cook(80) \rightarrow Program(76))$,
	$(Idle(68) \rightarrow Cook(80)), (Cook(80) \rightarrow Idle(68))$
$Not-Time(42)$	$[S : ()] : (Counter(45) \rightarrow Power(49)), (Power(49) \rightarrow Counter(45))$
$Set-Time(108)$	$[S : Setting(111)]$:
	$(Setting(111) \rightarrow Error(115)), (Setting(111) \rightarrow Setting(111))$
$Program(119)$	$[S : Setting(122)]$:
	$(Setting(122) \rightarrow Power(126)), (Power(126) \rightarrow Setting(122))$,
	$(Setting(122) \rightarrow Setting(122))$
$Cook(130)$	$[S : RealCook(133)]$:
	$(RealCook(133) \rightarrow RealCook(133))$

autoradio.cte

Car Radio Example [Hel07]: The car radio consists of three main classifications, a CD Drive (*CD Fach*), a Tape Deck (*Cassetten Fach*) and a Control Unit (*Steuerung*). The CD Drive has three modes, which are Empty (*cd_leer*), Reading (*Titel einlesen*) and Loaded (*cd_voll*). The Tape Deck only has Empty (*t_leer*) and Loaded (*t_voll*). The Control Unit can either be On (*An*) or Off (*Aus*). If it is on, then there are three distinct playback options: *Radio Modus*, *Cassette Modus* and *CD Modus*. Each option features its own charateristics, there are 4 radio station keys, for example.

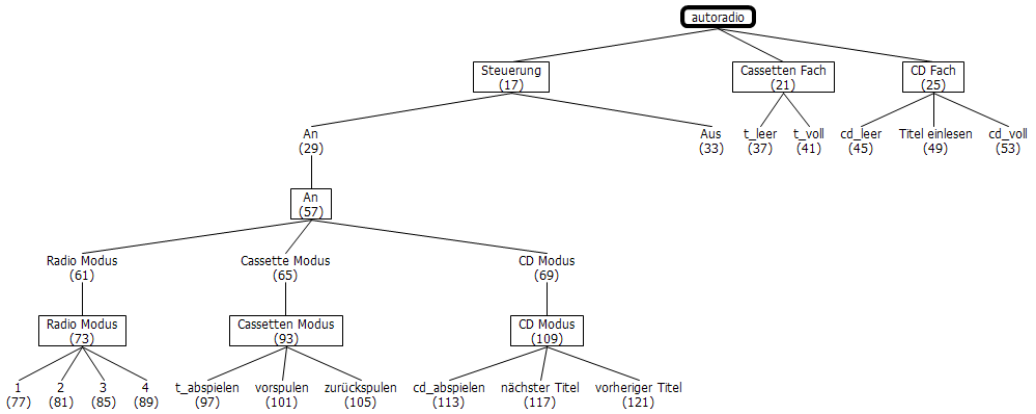

Figure A.3: Autoradio Example

$Steuerung(17)$ $[S : An(29)] :$
$(An(29) \to Aus(33)), (Aus(33) \to An(29))$

$CassettenFach(21)$ $[S : t\_leer(37)] :$
$(t\_leer(37) \to t\_voll(41)), (t\_voll(41) \to t\_leer(37))$

$CDFach(25)$ $[S : cd\_leer(45)] :$
$(cd\_leer(45) \to Titeleinlesen(49)),$
$(Titeleinlesen(49) \to Titeleinlesen(49)),$
$(Titeleinlesen(49) \to cd\_voll(53)), (cd\_voll(53) \to cd\_leer(45))$

$An(57)$ $[S : RadioModus(61)] :$
$(RadioModus(61) \to CassetteModus(65)),$
$(CassetteModus(65) \to RadioModus(61)),$
$(CassetteModus(65) \to RadioModus(61)),$
$(CassetteModus(65) \to RadioModus(61)),$
$(CassetteModus(65) \to CDModus(69)),$
$(CassetteModus(65) \to CDModus(69)),$
$(CDModus(69) \to RadioModus(61)),$
$(CDModus(69) \to RadioModus(61)),$
$(CDModus(69) \to RadioModus(61)),$
$(RadioModus(61) \to CDModus(69))$

$RadioModus(73)$ $[S : 1(77)] :$
$(1(77) \to 2(81)), (2(81) \to 1(77)), (2(81) \to 3(85)), (3(85) \to 2(81)),$
$(3(85) \to 4(89)), (4(89) \to 3(85)), (4(89) \to 1(77)), (1(77) \to 4(89))$

$CassettenModus(93)$ $[S : t\_abspielen(97)] :$
$(t\_abspielen(97) \to vorspulen(101)),$
$(vorspulen(101) \to t\_abspielen(97)),$
$(t\_abspielen(97) \to zurückspulen(105)),$
$(zurückspulen(105) \to t\_abspielen(97)),$
$(zurückspulen(105) \to t\_abspielen(97))$

$CDModus(109)$ $[S : cd\_abspielen(113)] :$
$(cd\_abspielen(113) \to nächsterTitel(117)),$
$(nächsterTitel(117) \to cd\_abspielen(113)),$
$(cd\_abspielen(113) \to vorherigerTitel(121)),$
$(vorherigerTitel(121) \to cd\_abspielen(113)),$
$(cd\_abspielen(113) \to nächsterTitel(117))$

citizen.cte

Citizen Watch [Har87]: The Citizen Watch was Harel's original example to introduce Statecharts in [Har87].

The watch can either be *alive* or *dead*. If it is alive, then it consists of six parellel activities, which are *main*, *alarm1-status*, *alarm2-status*, *chime-status*, *light*, and *power*. While the other five activities can be only *disabled* or *enabled*, basically, Main is divided into severeal further substates. They are *display* and *alarm-beep*. If the watch is on alarm-beep, then it is *alarm 1 beep*, *alarm 2 beep* or *both beep.*. If the watch is not on alarm beep, it is on *display*. The display can be one of *regular*, *wait*, *out* or *stopwatch*, which are then further refined.

We skip the rest of description and refer to the figures for further details.

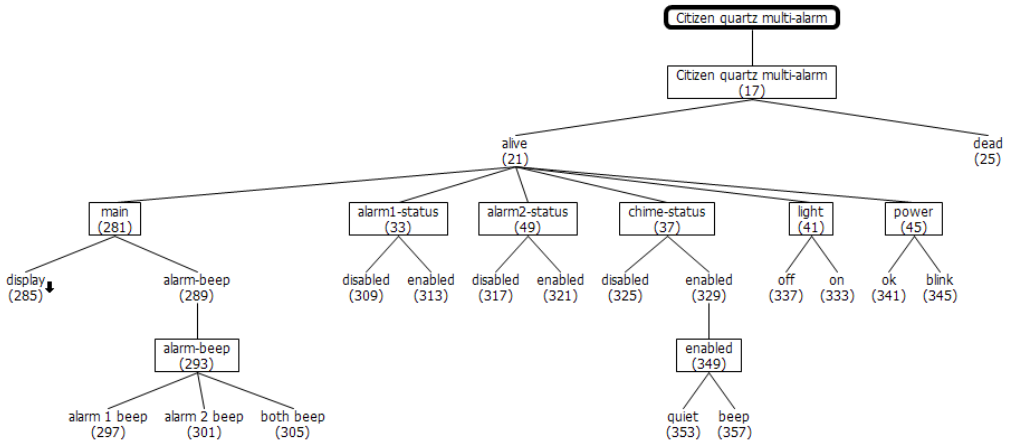

Figure A.4: Citizen Example

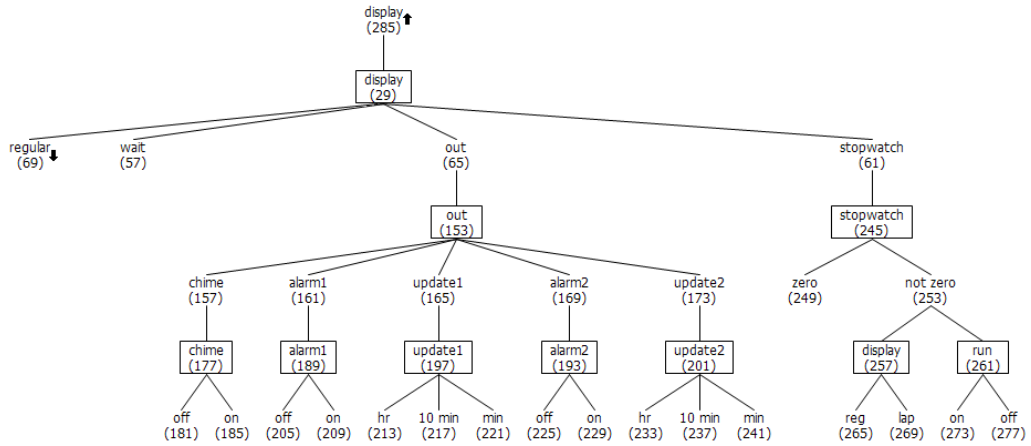

Figure A.5: Citizen Example

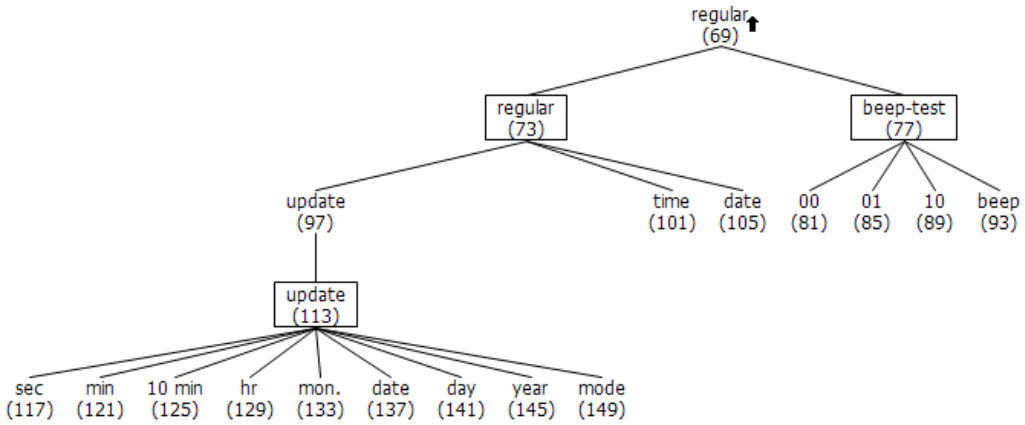

Figure A.6: Citizen Example

$Citizen(17)$	$[S : dead(25)]$:
	$(dead(25) \rightarrow alive(21)), (alive(21) \rightarrow dead(25))$
$main(281)$	$[S : display(285)]$:
	$(display(285) \rightarrow alarm1beep(297)), (display(285) \rightarrow alarm2beep(301)),$
	$(display(285) \rightarrow bothbeep(305)), (alarm-beep(289) \rightarrow display(285))$
$alarm1-status(33)$	$[S : disabled(309)]$:
	$(disabled(309) \rightarrow enabled(313)), (enabled(313) \rightarrow disabled(309))$
$alarm2-status(49)$	$[S : disabled(317)]$:
	$(disabled(317) \rightarrow enabled(321)), (enabled(321) \rightarrow disabled(317))$
$enabled(349)$	$[S : ()]$:
	$(quiet(353) \rightarrow beep(357)), (beep(357) \rightarrow quiet(353))$
$chime-status(37)$	$[S : disabled(325)]$:
	$(disabled(325) \rightarrow quiet(353)), (enabled(329) \rightarrow disabled(325))$
$light(41)$	$[S : off(337)]$:
	$(off(337) \rightarrow on(333)), (on(333) \rightarrow off(337))$
$power(45)$	$[S : ok(341)]$:
	$(ok(341) \rightarrow blink(345)), (blink(345) \rightarrow dead(25))$
$display(29)$	$[S : regular(69)]$:
	$(out(65) \rightarrow regular(69)), (stopwatch(61) \rightarrow regular(69)),$
	$(wait(57) \rightarrow sec(117))$

$regular(73)$ $[S : time(101)]$:
$(time(101) \rightarrow date(105)), (date(105) \rightarrow time(101)),$
$(time(101) \rightarrow wait(57)), (update(97) \rightarrow time(101)),$
$(time(101) \rightarrow alarm1(161))$

$beep-test(77)$ $[S : 00(81)]$:
$(00(81) \rightarrow 10(89)), (10(89) \rightarrow 00(81)), (00(81) \rightarrow 01(85)),$
$(01(85) \rightarrow 00(81)), (10(89) \rightarrow beep(93)), (beep(93) \rightarrow 10(89)),$
$(01(85) \rightarrow beep(93)), (beep(93) \rightarrow 01(85))$

$out(153)$ $[S : ()]$:
$(alarm1(161) \rightarrow alarm2(169)), (update2(173) \rightarrow alarm2(169)),$
$(alarm2(169) \rightarrow chime(157)), (alarm2(169) \rightarrow hr(233)),$
$(update1(165) \rightarrow alarm1(161)), (alarm1(161) \rightarrow hr(213)),$
$(chime(157) \rightarrow stopwatch(61))$

$stopwatch(245)$ $[S : zero(249)]$:
$(zero(249) \rightarrow notzero(253)), (zero(249) \rightarrow reg(265))$

$update(113)$ $[S : ()]$:
$(sec(117) \rightarrow min(121)), (min(121) \rightarrow 10min(125)),$
$(10min(125) \rightarrow hr(129)), (hr(129) \rightarrow mon.(133)),$
$(mon.(133) \rightarrow date(137)), (date(137) \rightarrow day(141)),$
$(day(141) \rightarrow year(145)), (year(145) \rightarrow mode(149)),$
$(mode(149) \rightarrow time(101))$

$chime(177)$ $[S : off(181)]$:
$(off(181) \rightarrow on(185)), (on(185) \rightarrow off(181))$

$alarm1(189)$ $[S : off(205)]$:
$(off(205) \rightarrow on(209)), (on(209) \rightarrow off(205))$

$update1(197)$ $[S : ()]$:
$(hr(213) \rightarrow 10min(217)), (10min(217) \rightarrow min(221)),$
$(min(221) \rightarrow alarm1(161))$

$alarm2(193)$ $[S : off(225)]$:
$(off(225) \rightarrow on(229)), (on(229) \rightarrow off(225))$

$update2(201)$ $[S : ()]$:
$(hr(233) \rightarrow 10min(237)), (10min(237) \rightarrow min(241)),$
$(min(241) \rightarrow alarm2(169))$

$display(257)$ $[S : reg(265)]$:
$(reg(265) \rightarrow lap(269)), (lap(269) \rightarrow reg(265)), (reg(265) \rightarrow zero(249))$

$run(261)$ $[S : on(273)]$:
$(on(273) \rightarrow off(277)), (off(277) \rightarrow on(273))$

coffee.cte

Coffee Machine Example: IBM Rhapsody features this example of a commercial coffee machine. It consists of the actual *CoffeeMachine*, a *Display* and a *Boiler*. All three of them can be either *off* or *on* independent of each other.

If the CoffeeMachine is on, its *Main* state is either *idle* or *working* while the *Power* state is always in its *working* mode. If the machine is idle, it can be in *waitMaintenance* or in *readyForCoin*. When a coin is supplied, the machine changes to its working state and produces a cup of coffee step by step: *fillCoffee*, *pressurize*, *fillWater*, *fillMilk*, *fillCup*, and *cupReady*.

The display shows progress messages either *permanent* or *rotating*.

The boiler has a temperatur sensor and cam *boilWater* or *keepTemp*.

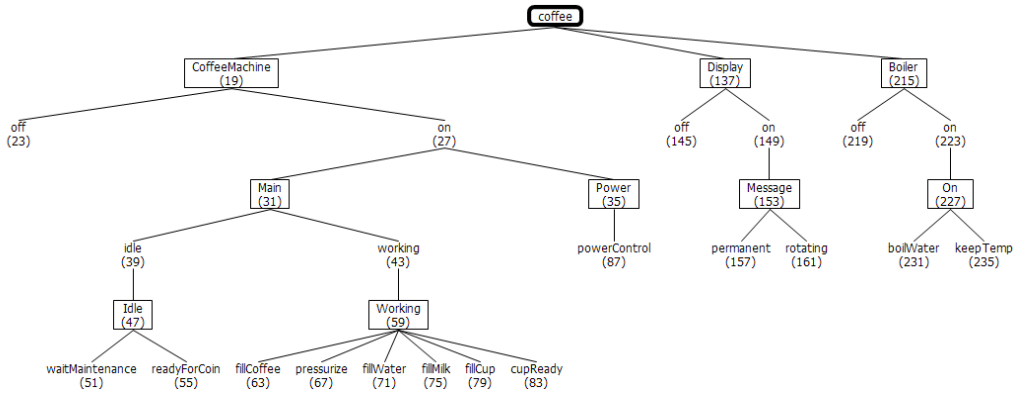

Figure A.7: Coffee Machine Example

$$
\begin{aligned}
\mathit{CoffeeMachine}(19) \quad & [S : \mathit{off}(23)] : \\
& (\mathit{off}(23) \rightarrow \mathit{on}(27)), (\mathit{on}(27) \rightarrow \mathit{off}(23)) \\
\mathit{Display}(137) \quad & [S : \mathit{off}(145)] : \\
& (\mathit{off}(145) \rightarrow \mathit{on}(149)), (\mathit{on}(149) \rightarrow \mathit{off}(145)), \\
& (\mathit{off}(145) \rightarrow \mathit{off}(145)) \\
\mathit{Boiler}(215) \quad & [S : \mathit{off}(219)] : \\
& (\mathit{off}(219) \rightarrow \mathit{off}(219)), (\mathit{off}(219) \rightarrow \mathit{on}(223)), \\
& (\mathit{on}(223) \rightarrow \mathit{off}(219)) \\
\mathit{Main}(31) \quad & [S : \mathit{idle}(39)] : \\
\\
\mathit{Power}(35) \quad & [S : \mathit{powerControl}(87)] :
\end{aligned}
$$

Message(153)	[S : *permanent*(157)] :
	(*permanent*(157) → *rotating*(161)),
	(*rotating*(161) → *permanent*(157)),
	(*rotating*(161) → *rotating*(161))
On(227)	[S : *boilWater*(231)] :
	(*boilWater*(231) → *keepTemp*(235)),
	(*keepTemp*(235) → *boilWater*(231))
Idle(47)	[S : *waitMaintenance*(51)] :
	(*waitMaintenance*(51) → *readyForCoin*(55)),
	(*readyForCoin*(55) → *waitMaintenance*(51)),
	(*readyForCoin*(55) → *working*(43))
Working(59)	[S : *fillCoffee*(63)] :
	(*fillCoffee*(63) → *fillCoffee*(63)),
	(*fillCoffee*(63) → *pressurize*(67)),
	(*pressurize*(67) → *pressurize*(67)),
	(*pressurize*(67) → *fillWater*(71)),
	(*fillWater*(71) → *fillWater*(71)),
	(*fillWater*(71) → *fillMilk*(75)),
	(*fillMilk*(75) → *fillMilk*(75)),
	(*fillMilk*(75) → *fillCup*(79)),
	(*fillCup*(79) → *fillCup*(79)),
	(*fillCup*(79) → *cupReady*(83)),
	(*cupReady*(83) → *idle*(39))

communication.cte

Communication Example: IBM Rhapsody features this example of a communication software. The *State* of the software can either be *Disabled* or *Enabled*. If it is Enabled, it can be *Idle*, *Receiving* or *Transmitting*. In Receiving, the software is *Waiting_for_Byte*, *Validating_Command* or *Processing*. In Transmitting, the software is *Waiting_for_Timeout* or *Waiting_for_Transmit_Timeout*.

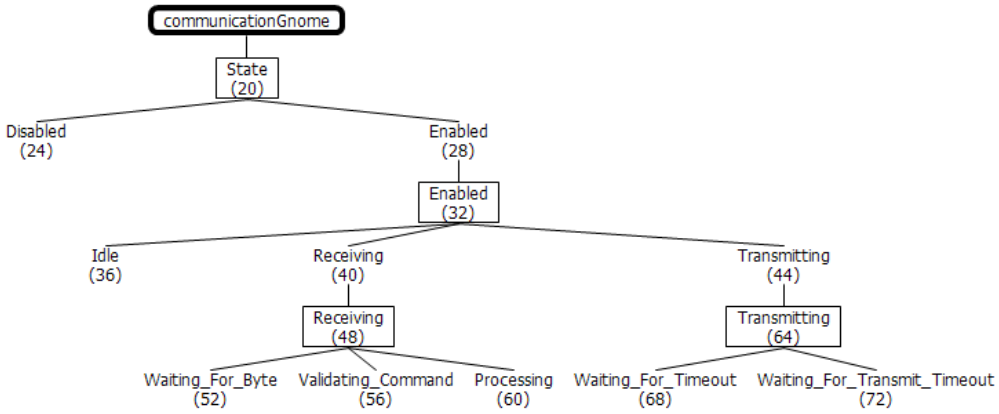

Figure A.8: Communication Example

$$
\begin{aligned}
State(20) \quad & [S : Disabled(24)] : \\
& (Disabled(24) \rightarrow Idle(36)), (Enabled(28) \rightarrow Disabled(24)) \\
Enabled(32) \quad & [S : Idle(36)] : \\
& (Idle(36) \rightarrow Recieving(40)), (Idle(36) \rightarrow Transmitting(44)) \\
Recieving(48) \quad & [S : Waiting\_For\_Byte(52)] : \\
& (Waiting\_For\_Byte(52) \rightarrow Waiting\_For\_Byte(52)), \\
& (Waiting\_For\_Byte(52) \rightarrow Validating\_Command(56)), \\
& (Validating\_Command(56) \rightarrow Processing(60)), \\
& (Validating\_Command(56) \rightarrow Transmitting(44)), \\
& (Processing(60) \rightarrow Transmitting(44)) \\
Transmitting(64) \quad & [S : Waiting\_For\_Timeout(68)] : \\
& (Waiting\_For\_Timeout(68) \rightarrow Waiting\_For\_Timeout(68)), \\
& (Waiting\_For\_Timeout(68) \rightarrow Waiting\_For\_Transmit\_Timeout(72)), \\
& (Waiting\_For\_Transmit\_Timeout(72) \rightarrow Idle(36))
\end{aligned}
$$

elevator.cte

Elevator Example: IBM Rhapsody features this example of an elevator. The elevator operation is defined by an *Action*, *Direction*, *Door* and *Building*. Actions can be *wait*, *changeDirection*, *moving*, *openingDoor* and *closingDoor*. Directions are *goingUp* and *goingDown*. The door is either *closed*, *opening*, *open*, or *closing*. The building is either *normal* or *simulated*.

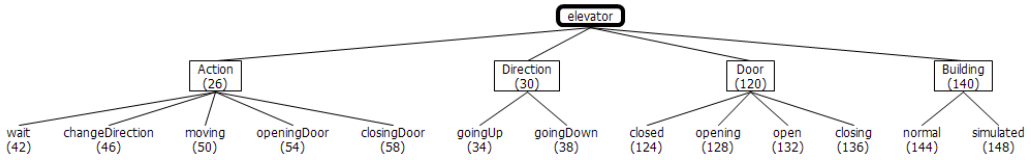

Figure A.9: Elevator Example

$$
\begin{aligned}
Action(26) \quad & [S : wait(42)] : \\
& (wait(42) \rightarrow changeDirection(46)), \\
& (changeDirection(46) \rightarrow wait(42)), \\
& (wait(42) \rightarrow moving(50)), \\
& (wait(42) \rightarrow openingDoor(54)), \\
& (moving(50) \rightarrow openingDoor(54)), \\
& (openingDoor(54) \rightarrow closingDoor(58)), \\
& (closingDoor(58) \rightarrow wait(42)), (moving(50) \rightarrow moving(50)) \\
Direction(30) \quad & [S : goingUp(34)] : \\
& (goingUp(34) \rightarrow goingDown(38)), (goingDown(38) \rightarrow goingUp(34)) \\
Door(120) \quad & [S : closed(124)] : \\
& (closed(124) \rightarrow opening(128)), (opening(128) \rightarrow open(132)), \\
& (open(132) \rightarrow closing(136)), (closing(136) \rightarrow closed(124)) \\
Building(140) \quad & [S : normal(144)] : \\
& (normal(144) \rightarrow normal(144)), (normal(144) \rightarrow simulated(148)), \\
& (simulated(148) \rightarrow simulated(148)), \\
& (simulated(148) \rightarrow normal(144))
\end{aligned}
$$

tetris.cte

Tetris Example: IBM Rhapsody features this example of a Tetris game. The game can either be *ready*, *startingGame*, *newPiece*, *runningGame*, *paused*, *removePlace*, or *gameOver*. RunningGame is further refined into *startUp*, *controlling*, *dropping*, and *dropped*. There are no parallel activities involved in this example.

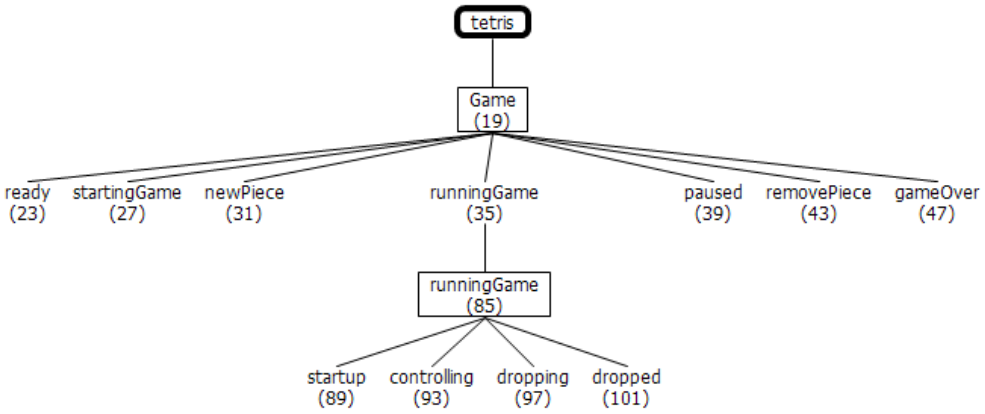

Figure A.10: Tetris Example

$$
\begin{aligned}
Game(19) \quad & [S : ready(23)] : \\
& (ready(23) \rightarrow startingGame(27)), \\
& (startingGame(27) \rightarrow newPiece(31)), \\
& (newPiece(31) \rightarrow runningGame(35)), \\
& (runningGame(35) \rightarrow paused(39)), \\
& (paused(39) \rightarrow runningGame(35)), \\
& (removePiece(43) \rightarrow newPiece(31)), \\
& (removePiece(43) \rightarrow newPiece(31)), \\
& (paused(39) \rightarrow gameOver(47)), \\
& (gameOver(47) \rightarrow startingGame(27)), \\
& (runningGame(35) \rightarrow gameOver(47)) \\
runningGame(85) \quad & [S : startup(89)] : \\
& (startup(89) \rightarrow controlling(93)), \\
& (controlling(93) \rightarrow dropping(97)), \\
& (controlling(93) \rightarrow dropped(101)), \\
& (dropping(97) \rightarrow dropped(101)), \\
& (startup(89) \rightarrow gameOver(47)), \\
& (dropping(97) \rightarrow dropping(97)), \\
& (controlling(93) \rightarrow controlling(93)), \\
& (dropped(101) \rightarrow controlling(93)), \\
& (dropped(101) \rightarrow removePiece(43))
\end{aligned}
$$

moore.cte

Moore Example: Matlab Simulink Stateflow features this example. States are *s0*, *s1*, *s12*, *s121*, *s1123*. There are no parallel activities involved in this example.

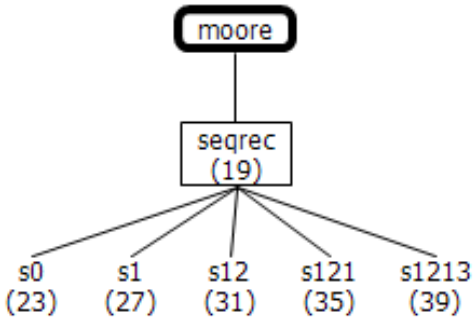

Figure A.11: Moore Example

$seqrec(19)$ $[S : s0(23)]$:
$(s0(23) \rightarrow s1(27)), (s1(27) \rightarrow s0(23)), (s1(27) \rightarrow s12(31)),$
$(s12(31) \rightarrow s0(23)), (s12(31) \rightarrow s121(35)), (s121(35) \rightarrow s12(31)),$
$(s121(35) \rightarrow s0(23)), (s121(35) \rightarrow s1(27)), (s121(35) \rightarrow s1213(39)),$
$(s1213(39) \rightarrow s0(23)), (s1213(39) \rightarrow s1(27))$

fuel.cte

Fuel Example: Matlab Simulink Stateflow features this example. Parellel states consist of *O2*, *Pressure*, *Throttle*, *Speed*, *Fail* and *Fueling_Mode*. Oxygen *O2* can be *O2_warmup*, *O2_normaln* and *O2_failure*. All three Pressure, Throttle and Speed can be *normal* and *fail*. The Fail state indicates the number of fails, it can be *None*, *One* and *Multi*. Multi is then further refined into *Two*, *Three* and *Four*. The Fueling Mode is one of *Running* and *Fuel_Disabled*. If it is Running, it is one of *Low_Emissions* (with *Warmup* and *Normal*) and *Rich_Mixture* (with only *Single_Failure*). Fuel_Disabled can be due to *Overspeed* or *Shutdown*.

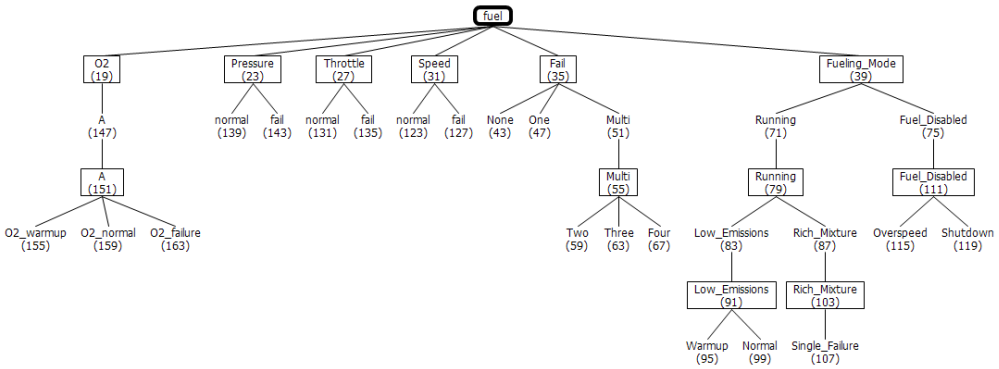

Figure A.12: Fuel Example

$$
\begin{aligned}
O2(19) \quad & [S : A(147)] : \\
\\
Pressure(23) \quad & [S : normal(139)] : \\
& (normal(139) \rightarrow fail(143)), (fail(143) \rightarrow normal(139)) \\
Throttle(27) \quad & [S : normal(131)] : \\
& (normal(131) \rightarrow fail(135)), (fail(135) \rightarrow normal(131)) \\
Speed(31) \quad & [S : normal(123)] : \\
& (normal(123) \rightarrow fail(127)), (fail(127) \rightarrow normal(123)) \\
Fail(35) \quad & [S : None(43)] : \\
& (None(43) \rightarrow One(47)), (One(47) \rightarrow None(43)), \\
& (One(47) \rightarrow Two(59)) \\
Fueling\_Mode(39) \quad & [S : Running(71)] : \\
& (Running(71) \rightarrow Overspeed(115)), \\
& (Running(71) \rightarrow Shutdown(119)) \\
A(151) \quad & [S : O2\_warmup(155)] : \\
& (O2\_warmup(155) \rightarrow O2\_normal(159)), \\
& (O2\_normal(159) \rightarrow O2\_failure(163)), \\
& (O2\_failure(163) \rightarrow O2\_normal(159)) \\
Multi(55) \quad & [S : ()] : \\
& (Two(59) \rightarrow Three(63)), (Three(63) \rightarrow Two(59)), \\
& (Three(63) \rightarrow Four(67)), (Four(67) \rightarrow Three(63)), \\
& (Two(59) \rightarrow One(47)) \\
Running(79) \quad & [S : Low\_Emissions(83)] :
\end{aligned}
$$

Fuel_Disabled(111)	[S : ()] :
	(*Overspeed*(115) → *Shutdown*(119)),
	(*Overspeed*(115) → *Running*(71)),
	(*Shutdown*(119) → *Rich_Mixture*(87))
Low_Emissions(91)	[S : *Warmup*(95)] :
	(*Warmup*(95) → *Normal*(99)),
	(*Warmup*(95) → *Single_Failure*(107)),
	(*Normal*(99) → *Single_Failure*(107))
Rich_Mixture(103)	[S : *Single_Failure*(107)] :
	(*Single_Failure*(107) → *Normal*(99))

transmission.cte

Transmission Example: Matlab Simulink Stateflow features this example of a Gear Box. The *gear_state* can be one of *first*, *second*, *third*, or *forth*. The *selection_state* can be *steady_state*, *downshifting* or *upshifting*.

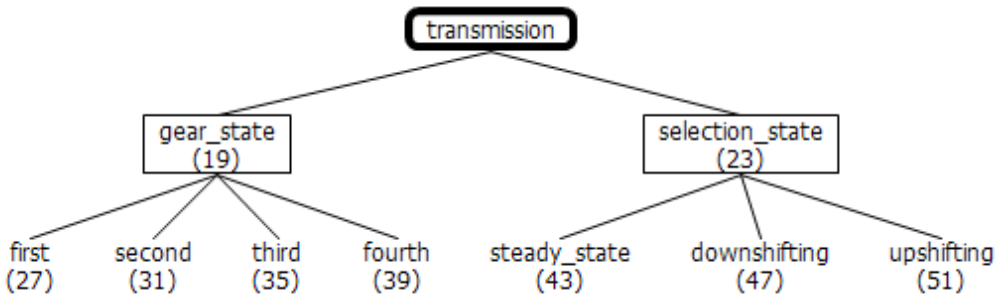

Figure A.13: Transmission Example

gear_state(19)	$[S : first(27)] :$
	$(first(27) \rightarrow second(31)), (second(31) \rightarrow third(35)),$
	$(third(35) \rightarrow fourth(39)), (fourth(39) \rightarrow third(35)),$
	$(third(35) \rightarrow second(31)), (second(31) \rightarrow first(27))$
selection_state(23)	$[S : steady\_state(43)] :$
	$(steady\_state(43) \rightarrow downshifting(47)),$
	$(downshifting(47) \rightarrow steady\_state(43)),$
	$(downshifting(47) \rightarrow steady\_state(43)),$
	$(steady\_state(43) \rightarrow upshifting(51)),$
	$(upshifting(51) \rightarrow steady\_state(43)),$
	$(upshifting(51) \rightarrow steady\_state(43))$

aircraft.cte

Aircraft Example: Matlab Simulink Stateflow features this example of a redundant aircraft sub system. One component exists four times: *LO*, *RO*, *LI*, and *RI*. Each item can be *L1* or *isolated*. If it is L1, then it is one of *passive*, *standby*, *active*, or *off*.

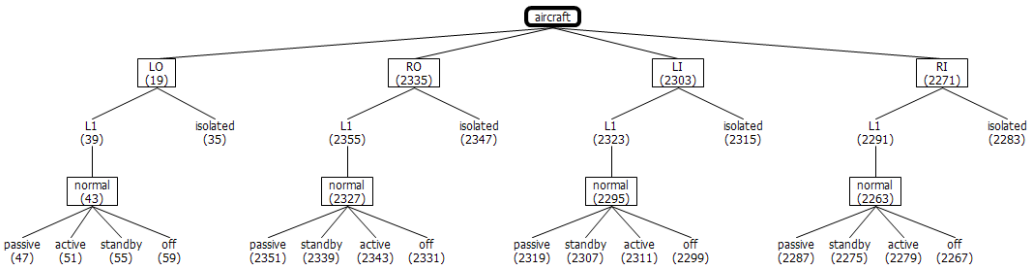

Figure A.14: Aircraft Example

$LO(19)$ $[S : L1(39)] :$
$(L1(39) \rightarrow isolated(35))$

$RO(2335)$ $[S : L1(2355)] :$
$(L1(2355) \rightarrow isolated(2347))$

$LI(2303)$ $[S : L1(2323)] :$
$(L1(2323) \rightarrow isolated(2315))$

$RI(2271)$ $[S : L1(2291)] :$
$(L1(2291) \rightarrow isolated(2283))$

$normal(43)$ $[S : passive(47)] :$
$(passive(47) \rightarrow active(51)), (passive(47) \rightarrow standby(55)),$
$(active(51) \rightarrow standby(55), (standby(55) \rightarrow active(51))$

$normal(2327)$ $[S : passive(2351)] :$
$(passive(2351) \rightarrow standby(2339)), (passive(2351) \rightarrow active(2343)),$
$(standby(2339) \rightarrow active(2343)), (active(2343) \rightarrow standby(2339))$

$normal(2295)$ $[S : passive(2319)] :$
$(passive(2319) \rightarrow standby(2307)), (passive(2319) \rightarrow active(2311)),$
$(standby(2307) \rightarrow active(2311)), (active(2311) \rightarrow standby(2307))$

$normal(2263)$ $[S : passive(2287)] :$
$(passive(2287) \rightarrow standby(2275)), (standby(2275) \rightarrow active(2279)),$
$(passive(2287) \rightarrow active(2279)), (active(2279) \rightarrow standby(2275))$

Versicherung

Ich erkläre, dass ich die vorliegende Dissertation selbständig und eigenhändig angefertigt und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe.

Ich erkläre weiterhin, dass die Dissertation bisher nicht in dieser oder anderer Form in einem anderen Prüfungsverfahren vorgelegen hat.

_____

Berlin, den