Techniques for Automatic Generation of Tests from Programs and Specifications

by

Jon Edvardsson
Abstract

SOFTWARE TESTING IS COMPLEX and time consuming. One way to reduce the effort associated with testing is to generate test data automatically. This thesis is divided into three parts. In the first part a mixed-integer constraint solver developed by Gupta et. al is studied. The solver, referred to as the Unified Numerical Approach (UNA), is an important part of their generator and it is responsible for solving equation systems that correspond to the program path currently under test.

In this thesis it is shown that, in contrast to traditional optimization methods, the UNA is not bounded by the size of the solved equation system. Instead, it depends on how the system is composed. That is, even for very simple systems consisting of one variable we can easily get more than a thousand iterations. It is also shown that the UNA is not complete, that is, it does not always find a mixed-integer solution when there is one. It is found that a better approach is to use a traditional optimization method, like the simplex method in combination with branch-and-bound and/or a cutting-plane algorithm as a constraint solver.

The second part explores a specification-based approach for generating tests developed by Meudec. Tests are generated by partitioning the specification input domain into a set of subdomains using a rule-based automatic partitioning strategy. An important step of Meudec’s method is to reduce the number of generated subdomains and find a minimal partition. This thesis shows that Meudec’s minimal partition algorithm is incorrect. Furthermore, two new efficient alternative algorithms are developed. In addition, an algorithm for finding the upper and lower bound on the number of subdomains in a partition is also presented.

Finally, in the third part, two different designs of automatic testing tools are studied. The first tool uses a specification as an oracle. The second tool, on the other hand, uses a reference program. The fault-detection effectiveness of the tools is evaluated using both randomly and systematically generated inputs.

This work has been supported ECSEL (Excellence Center in Computer Science and Systems Engineering in Linköping) and NUTEK (the Swedish National Board for Industrial and Technical Development)
Writing my doctoral thesis has been like when I ran my first marathon. It required serious effort just to get into shape to start the race. Then, several times along the road, I got tired and my body wanted me to stop. But everywhere I looked there were people cheering and supporting me and the other runners. Their support and my stubbornness kept forcing me forward, sometimes running, sometimes walking. Finally, when passing the finishing line all I felt was pure joy.

Now, when I am putting the last words to my doctoral thesis I would like to express my gratitude to all the people who have supported me along the way. First of all, I would like to thank my supervisor Professor Mariam Kamkar for all her support and guidance during my work. Not only has she spent much time discussing research plans and commenting and improving paper drafts, but most importantly, she has encouraged me to continue my work when things have been most difficult. For this I will always be grateful.

Also, I would like to express my gratitude to my second supervisor, Professor Maud Göthe-Lundgren, for her help in verifying the proofs of part I of this thesis. I am also thankful for her and Andreas Westerlund’s help with my work on the Path-cover Method.

Special thanks also go to my brother Tor, Jens Gustavsson, Carl Cederberg, John Wilander, and Johan Åberg for putting up with my endless monologues regarding my work. I would like to thank Brittany Shahmehri for proofreading my thesis. A special thanks to all my colleagues at PELAB (Programming Environment Laboratory) and IDA (Department of Computer and Information Science) for contributing to an excellent atmosphere where interesting and entertaining coffee-break discussions play a valuable role.

Last, but not least, I thank my family for all the support they have given me. Especially Anna for being such a great wife and friend, not to mention also mother of our wonderful children Algot and Rakel.

Jon Edvardsson
Linköping, May 2006
# Contents

1 Introduction
   1.1 Thesis Overview ........................................... 2
   1.2 Contributions .................................................. 4
   1.3 Publications .................................................... 4
      1.3.1 To be Submitted ........................................... 5

I Program-based Test Data Generation 7

2 Introduction to Program-based Test Data Generation 9
   2.1 Constraint Generator ........................................... 13
      2.1.1 Symbolic Execution ........................................ 15
      2.1.2 Actual Execution .......................................... 16
      2.1.3 Hybrid Approach ........................................... 17
      2.1.4 Goal-oriented vs. Path-oriented ............................ 18
      2.1.5 Syntax-oriented ............................................ 18
   2.2 Constraint Solver ................................................ 19

3 Introduction to the Unified Numerical Approach 23
   3.1 Basic Concepts and Notations .................................. 24
      3.1.1 Equation Systems .......................................... 25
   3.2 Introduction to UNA ............................................. 26
   3.3 UNA Algorithm ................................................... 30
      3.3.1 Implementation ............................................ 31

4 Complexity and Incompleteness of UNA 33
   4.1 Complexity of UNA .............................................. 33
   4.2 Incompleteness of UNA .......................................... 37
      4.2.1 Why Incomplete – Intuitive Explanation .................. 41
      4.2.2 Why Incomplete – Mathematical Explanation ............ 42
      4.2.3 Examples .................................................... 44
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 New Approach</td>
</tr>
<tr>
<td>5.1 Simplex vs. UNA</td>
</tr>
<tr>
<td>5.2 Benchmark</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
</tr>
<tr>
<td>6.1 Future work</td>
</tr>
<tr>
<td>II Specification-based Test Data Generation</td>
</tr>
<tr>
<td>7 Introduction to Specification-based TDG</td>
</tr>
<tr>
<td>7.1 Specification Languages</td>
</tr>
<tr>
<td>7.1.1 Styles of Specification Languages</td>
</tr>
<tr>
<td>7.2 Adequacy Criteria for Specifications</td>
</tr>
<tr>
<td>8 Partition Testing</td>
</tr>
<tr>
<td>8.1 Systematic Partitioning</td>
</tr>
<tr>
<td>8.2 Minimal Partitions</td>
</tr>
<tr>
<td>9 Meudec's Minimal Partition Method</td>
</tr>
<tr>
<td>9.1 Naive Approach</td>
</tr>
<tr>
<td>9.2 Graph Approach</td>
</tr>
<tr>
<td>9.3 Analysis of Meudec's Minimal Partition Algorithm</td>
</tr>
<tr>
<td>10 Tree-based Partial Expansion</td>
</tr>
<tr>
<td>10.1 Vertex Merging</td>
</tr>
<tr>
<td>10.2 Proof of Concept</td>
</tr>
<tr>
<td>10.3 The Tree Rules</td>
</tr>
<tr>
<td>10.4 Rationale</td>
</tr>
<tr>
<td>11 Set-cover Method</td>
</tr>
<tr>
<td>11.1 Set Covers</td>
</tr>
<tr>
<td>11.2 Column Generation</td>
</tr>
<tr>
<td>11.2.1 Preliminaries</td>
</tr>
<tr>
<td>11.2.2 Auxiliary Problem – Longest Path</td>
</tr>
<tr>
<td>11.3 Performance</td>
</tr>
<tr>
<td>11.3.1 Time Complexity</td>
</tr>
<tr>
<td>11.3.2 Space Complexity</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

— Thou shalt test, test, and test again.

9th commandment of formal methods [1]

Even though testing can only show the presence of faults and not the absence, it is still an important phase of program development and cannot be ruled out. There are many places in the development cycle where faults can be mistakenly introduced or let through, for instance, in requirements, analysis, design, implementation and even in testing. Some formal methods, such as VDM and B, enforce program correctness by requiring the writing of a formal specification for the actual program. Afterwards, in a series of refinement steps it is converted to an executable program. Since each step of refinement is proved to preserve correctness, the final program should also be correct. Even in this case, we must still perform testing since the program will most likely interact with other software or hardware components that may not be correct. For instance, if the program is compiled on a faulty compiler the resulting binary executable will not be correct.

In many software projects it is important that the development is as flexible as possible, while still being rigorous. This is to better withstand sudden and unexpected changes from customers. If too much effort is spent on the early phases of program development there may never be a program implemented before the market has changed.

Extreme programming (XP) [2] is an agile method inspired by this idea of flexible and rigorous development. Agile methods [3] is a term used to denote a group of development methods that, among other things, are incremental and test driven.

Flexibility of XP is enforced by having short iteration cycles with clear goals. Here, a goal would be a single system feature. Rigorousness in XP
is ensured by a strict testing policy: always write test cases before writing code. Writing test cases serves both as a design tool and a testing tool. By specifying test cases programmers get a feel for how the program should be designed and implemented. Test driven or not, we should test a program as early and often as possible [1, 4]. With this state of mind, not only will we find bugs early, but we might even prevent them from occurring.

Testing is expensive. At least 50% of the effort put into producing a working program comes from testing [4]. The earlier faults can be found, the more money can be saved on testing and maintenance, and thus we can cut costs or spend more on actual development. One way of reducing the testing effort is to automatically generate test data. In an article by Paradkar et al. [5] the testing phase is divided into the following steps:

1. Generation of test constraints, that is, the conditions that input variables need to satisfy, for a given adequacy criterion.

2. Generation of test data satisfying these constraints.

3. Test execution.

4. Output verification.

While execution is simple to automate, at least when considering unit testing, the other steps are more difficult.

1.1 Thesis Overview

Part I introduces the area of program-based test data generation (TDG) (step 1 and 2) and presents an in-depth study of a constraint solving technique known as the Universal Numerical Approach (UNA) (step 2). This solver developed by Gupta et al. has been integrated in their program-based TDG [6–8]. Their generator is based on novel technique which approximates path constraints by a linear equation system, which in turn is solved by the UNA solver. However, this work shows that the UNA has some undesirable properties. First, it is unbounded, meaning that there is no upper bound on the number of iterations needed to find a solution. Secondly, it is incomplete, that is, it does not always find a solution even if there is one. Therefore, this work describes a better approach based on the classical simplex method. The two techniques are evaluated in an experiment.
In part II specification-based TDG is explored. Specifically, a thorough analysis of an automatic partition testing technique developed by Christophe Meudec is conducted. In his dissertation [9] a strategy for automatically deriving specification-based tests is presented. He describes an automatic partition testing process, which in a series of steps divides the input domain of a specification into a set of subdomains, a partition, that is sampled for test data.

The advantages of Meudec’s strategy are many. Deriving test data from a specification allows testing for unimplemented functionality, which cannot be done through program-based tests. The given specification can be used by an automatic oracle to validate the test result (step 4). Furthermore, instead of sampling the resulting partition, it can be used as an adequacy criterion for tests derived from another source, for instance, a test generated randomly or from the corresponding program.

Before this technique can actually be implemented many issues must be dealt with. Perhaps most importantly, the generated number of subdomains can assume enormous proportions. The problem is the equivalent of the exploding number of paths in testing based on path coverage. Meudec points out that many of the generated subdomains can be considered redundant and need not, therefore, be part of the complete partition. He formalizes these ideas in a definition of a so-called minimal partition and develops algorithm to find such a partition.

However, the work presented in the beginning of part II shows that Meudec’s algorithm is incorrect, that is, it does not always produce a minimal partition. This problem is more difficult than Meudec first anticipated.

Therefore, the rest of part II is dedicated to describing two alternative solutions for finding a minimal partition: set-cover method and path-cover method, both based on integer programming (IP). Benefits of defining the minimal partition problem as an IP problem is that there are well-known techniques to solve such problems.

Both methods require that the partition to minimize has certain properties, that is, dependence-free and singleton. Therefore a large portion of part II is spent on constructing an algorithm to generate such partitions from an initial partition that is not dependence-free. The technique used is referred to as partial expansion.

In part III, which is the last part of this thesis, two types of designs for automatic testing tools are studied. The first tool uses a specification as oracle. The second tool, on the other hand, uses a reference program. The fault-detection effectiveness of the tools are evaluated using both randomly and systematically generated inputs.
Chapter 1 Introduction

1.2 Contributions

- The UNA solver is shown to be unbounded and incomplete.
- A better approach based on the simplex method is suggested. The two approaches (Simplex and UNA) are compared experimentally.
- Meudec’s partition-minimization algorithm is proved to be incorrect.
- An algorithm form removing partition dependence is constructed.
- Two alternative methods for finding minimal partitions from independence partitions are presented.
- An algorithm to determine the upper and lower bound of the number of subdomains in a partition.
- An experimental study of the design of automatic testing tools.
- Structural coverage on specifications is shown to generate subdomains that are not desired in terms of good inputs.
- A serial design of a specification-based testing tool cannot find all types of faults.
- Random testing has high fault-detection effectiveness.

1.3 Publications

This thesis is based on the following publications of mine.


1.3.1 To be Submitted


Part I

Program-based Test Data Generation
Chapter 2
Introduction to Program-based Test Data Generation

— Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Dijkstra

The simplest form of automatic test data generation, also called input generation, is random testing. Instead of systematically selecting suitable test cases, input values for the program being tested are selected randomly. The simplicity of random generators is of great advantage. They do not incur the overhead of considering what parts of the program to test and finding input values that test exactly those parts. Furthermore, random generators work well with any input data type, since, ultimately, a data type such as integer, string, or heap is just a stream of bits. For example, a function having a string argument can be given a bit stream converted to a string.

Normally, an adequacy criterion is used in conjunction with a test data generator in order to know when to stop testing. For instance, we might say that at least 1000 test cases should be generated. However, the such a criterion, together with random testing, does not guarantee the quality of the generated test cases. For this reason, adequacy criteria are often correlated to the actual source code. For example, the criterion could be to reach statement coverage and therefore testing does not stop until all statements of the program have been executed at least once.

Intuitively, random testing could not be expected to perform well in terms of these stronger types of adequacy criteria, because it would have quite a low chance of finding parts of the program that contain semantically small faults, making high coverage difficult to achieve. A semantically small fault [16] is such a fault that is only revealed by a small
Listing 2.1: The probability of exercising \texttt{write(1)} is $1/n^2$, where $n$ is the number of integers in \texttt{int}.

```c
void foo(int a, int b) {
    if (a == b) then
        write(1);
    else
        write(2);
}
```

percentage of the program input. Consider the piece of code in figure 2.1. The probability of exercising the \texttt{write(1)} statement is $1/n^2$, where $n$ is the size of \texttt{int}, since in order to execute this statement variables $a$ and $b$ must be equal. We can easily imagine that generating even more complex structures than integers will give us even lower probability.

Partition testing, on the other hand, should be a more efficient type of testing in the sense that test data are systematically spread over the whole input domain. In partition testing the adequacy criterion is used to divide (partition) the program’s input domain into smaller parts called subdomains. From each subdomain one or more representatives are selected. The rationale is that each representative tests its subdomain equally well. For example, using statement coverage the above program can be partitioned into two subdomains \{ $a = b, a \neq b$ \}. A constraint on the input variables determines the \textit{shape} of a subdomain. Here, the first subdomain contains all inputs such that $a$ and $b$ are equal. Assigning values to the input variables such that the constraint holds gives us test data. In this case, selecting the two input assignments \{ $a = 1, b = 1$ \} and \{ $a = 1, b = 2$ \} is sufficient for statement coverage.

Partition testing may seem much better than random testing, but counterintuitively, results from Duran and Ntafos, Hamlet and Taylor, and Weyuker and Jeng shows that random testing is sometimes even better than partition testing in finding faults [17–19]. Later results, however, have shown that if subdomains are sampled proportionately to their size, partition testing is guaranteed to be at least as good as random testing [20].

For example, if a partition consists of the following 3 equal-sized subdomains \{ $\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ \}, selecting one candidate from each would guarantee that the test is at least as good as 3 random test cases. If a partition, on the
other hand, consists of non-equal-sized partitions

$$\left\{ \begin{array}{c} 1 \\
6 \\
3 \\
2 \end{array} \right\} = \left\{ \begin{array}{c} 1 \\
2 \\
3 \\
6 \\
6 \\
6 \end{array} \right\}$$

then selecting 1, 2, and 3 candidates respectively corresponds to a random test with 6 test cases. The result is a direct consequence of probability. However, despite the fact that partition testing is not entirely superior to random testing, it still ensures that the selected program parts (input domains) are exercised.

In figure 2.1 an overview of a typical architecture of a partition-oriented test data generator (TDG) is shown. Given a program, the TDG generates test data until the adequacy criteria is met. Adequacy is often stated as control-flow coverage, that is, the nodes or branches of the programs control-flow graph must be exercised in a particular way. For instance, statement coverage for the program given in figure 2.2 is achieved by a set of test data that traverses all nodes in the program’s control-flow graph. Similarly, branch-coverage is achieved when all edges are traversed. However, other adequacy criteria are applicable in this TDG model as well, for instance, data-flow coverage [4].

The TDG in figure 2.1 consists of a path selector, a constraint generator and a constraint solver. With respect to the given adequacy criterion, the path selector calculates what parts of the program are to be tested. Typically, a part can be thought of as a path through the program and therefore the term path is used in this text, even though it does not refer to a standard program path. The exact representation of a path is dependent on the adequacy criterion.

For each path, a constraint on the program’s input variables is generated. Each constraint is constructed such that when it holds, the corresponding path is traversed. A solution to the constraint, that is, an assignment to the input variables such that the constraint holds, represents the test data generated by the constraint solver.

The line between path selector and constraint generator (and constraint solver for that matter) is not always this distinct. However, separating their tasks is better from an explanatory point of view.

To restrict the scope of this thesis, the existence of a working path selection strategy it is assumed. For more information on path selection I recommend studying the path-prefix strategy proposed by Prather and Myers [21]. It ensures branch coverage modulo (relatively) infeasible paths. The strategy has been adopted by Chang et al. [22] in their heuristic approach for test data generation. Williams et al. [23] describes a similar method to generate $k$-path coverage.
Chapter 2 Introduction to Program-based Test Data Generation

Figure 2.1: Architecture of a test data generator.

```c
int triType(int a, int b, int c) {
    int type = PLAIN;
    if (a > b) swap(a, b);
    if (a > c) swap(a, c);
    if (b > c) swap(b, c);
    if (a == b) {
        if (b == c)
            type = EQUILATERAL;
        else
            type = ISOSCELES;
    }
    else if (b == c)
        type = ISOSCELES;
    return type;
}
```

Figure 2.2: A program that determines the type of a triangle and its corresponding control-flow graph.
Therefore, not considering path selection, the problem of automatic test data generation is as follows: given a program $p$ and a path $u$, generate input $x \in S$, where $S$ is the set of all inputs, such that when $p(x)$ is executed the path $u$ is traversed. This is achieved in two steps.

1. find the constraint for the path.
2. find a solution to the path constraint.

### 2.1 Constraint Generator

Consider the following problem: find a path constraint for path $p = \langle 1, 2, 3, 5, 6, 7, 8, 10, 13 \rangle$ for the program in figure 2.2. Before getting into details let us see what happens if we execute the program on the input $(5, 4, 4)$. Doing this we find that path $p$ is actually traversed. Naively, we may assume that a path constraint for $p$ corresponds to a conjunction of the constraints on the branches encountered when traversing the path in the control-flow graph.

$$P' = (a > b) \land (a \leq c) \land (b > c) \land (a = b) \land (b \neq c)$$

Let us assign $a = 5$, $b = 4$, and $c = 4$ and check whether $P'$ holds. Since $(5, 4, 4)$ does traverse the path $p$, then any path predicate corresponding to $p$ must hold for the corresponding input.

$$P' = (5 > 4) \land (5 \leq 4) \land (4 > 4) \land (5 = 4) \land (4 \neq 4)$$

Clearly we see that this is not the case. But why? When constructing the path constraint we ignored the effect of execution of the nodes of 1, 2, 6, and 10, which affects the values of variables in $P'$. Consequently, if the side effects are not taken into account, the constraint is invalid. For instance, assume that the program is executed on input $(5, 4, 4)$ and paused as the execution reaches node 7. Now, at this point we should expect $a = 4$ and $b = 5$, because before reaching node 7 the statement $\text{swap}(a, b)$ was executed, thus setting $a = 4$ and $b = 5$. In the naive constraint construction $\text{swap}(a, b)$ was not considered and therefore $a$ and $b$ still were equal to 5 and 4 respectively.
Chapter 2 Introduction to Program-based Test Data Generation

\[
\begin{array}{l}
1 \ (a > b) \quad \text{int type = PLAIN;}
3 \ (a \leq c) \quad \text{swap(a, b);}
5 \ (b > c)
7 \ (a = b) \quad \text{swap(b, c);}
8 \ (b \neq c)
13 \ \top \quad \text{type = ISOSCELES;}
\end{array}
\]

The above structure illustrates the data dependencies among branch predicates. Each row depends upon execution of itself as well as the previous rows. For instance, before checking whether \((a = b)\) in row 7 holds, the following must be executed: \(\text{int type = PLAIN; swap(a, b); swap(b, c);}\).

Thus, in order to adjust the branch predicates to take data dependence into account, one must perform the following steps. Start with the first row and execute its code. Update all succeeding rows (including the current condition) according to the side effects. Continue with the next row until all rows have been processed.

\[
\begin{array}{l}
1 \ (a > b)
3 \ (b \leq c)
5 \ (a > c)
7 \ (b = a) \quad \text{swap(a, c);} \\
8 \ (a \neq c)
13 \ \top \quad \text{type = ISOSCELES;}
\end{array}
\]

\[
\begin{array}{l}
1 \ (a > b)
3 \ (b \leq c)
5 \ (a > c)
7 \ (b = c)
8 \ (c \neq a)
13 \ \top
\end{array}
\]

Now each row corresponds to a branch predicate which is adjusted according to the execution of nodes 1, 2, 6, and 10. This gives us the new path constraint \(P = (a > b) \land (b \leq c) \land (a > c) \land (b = c) \land (c \neq a)\). If we again substitute \(a = 5\), \(b = 4\), and \(c = 4\) we see that \(P\) holds and therefore is a valid path constraint for path \(p\).

\[
P = (5 > 4) \land (4 \leq 4) \land (5 > 4) \land (4 = 4) \land (4 \neq 5)
\]

In order to find a valid path constraint two different techniques have been suggested in existing literature, namely: symbolic execution and
actual execution. Test data generators based on these techniques are also called static and dynamic TDG.

2.1.1 Symbolic Execution

Symbolic execution [24] is a popular approach for deriving path constraints. Executing a program in this way means that the program is manipulated symbolically, as if mathematical equations are rewritten, in order to get path constraints consisting of only input variables. For example, let \( a \) and \( b \) be input variables.

\[
c = a + b; \\
d = a - b; \\
e = c*d; \\
if (e > 5) 
    println(e);
\]

Using symbolic execution, the condition \( e > 5 \) in the above code will be transformed to \( a*a - b*b > 5 \). Assuming we let the constraint solver find a solution when this holds, we will have test data executing the path that ends with the `println(e)` statement.

In symbolic execution, arrays and pointers complicate the substitution since values of variables are not known. Note that arrays can be modeled using pointers. For example, \( a[5] \) can be written as \( *(a+5) \) in C.

Consider a condition statement comparing some array element indexed by an input dependent variable.

\[
\begin{align*}
\text{input}(i,j); \\
a[j] &= 2; \\
a[i] &= 0; \\
a[j] &= a[j] + 1; \\
\text{if} \ (a[j] == 3) \ldots
\end{align*}
\]

If \( i \) is equal to \( j \) then the value of \( a[j] \) in the `if` statement is 1, otherwise it is 3. Ramamoorthy et al. [25] propose an approach to solve this problem by creating a new instance of the assigned array when there is an ambiguous state. Whenever such ambiguity is resolved the array instances are merged. Of course, this approach suffers a large performance penalty.

Zhang reports on a tool [26, 27] capable of symbolic execution of programs containing pointer variables. The tool is not capable of handling memory deallocation. The main idea is to map pointer values to explicit addresses.
Generally a program is divided into functions and modules. When symbolically deriving a path constraint for a program calling other functions Ramamoorthy et al. [25] have proposed two solutions: either the brute force solution by inlining the called functions into the target, or by analyzing the called functions first and generating path predicates for those functions. But often, for instance in precompiled libraries, source code of a function or a module is not available and therefore a complete static analysis of the called functions on source level is not possible.

2.1.2 Actual Execution

An automatic test data generator described by Korel [28] is, on the contrary, based on actual execution of the program. Many of the problems associated with symbolic execution do not exist in actual execution, since values of variables are known at runtime and binary code is always available. Here the program is repeatedly executed until the correct path is traversed. By monitoring the program flow the test data generator can determine if the intended path was taken. If not, it backtracks to the point where the flow first deviated from the intended path. A search algorithm is used to improve the input values such that the intended branch is traversed. The test data generator continues in this manner until an input traversing the complete path is found or it has exceeded some user defined iteration limit.

In detail, instead of deriving a symbolic path constraint, this execution-based technique writes each branch constraint \( b_i \) on the form

\[
F_i(x) \text{ relop } 0,
\]

where \( \text{relop} \) is \(<, \leq, \text{ or } =\). For example, consider a program with the condition \( \text{if } (a \geq 10) \ldots \) on line 324. Then, with \( F(x) = a_{324} - 10 \), all input values \( x \) such that \( F(x) \leq 0 \) the condition \( a \geq 10 \) on line 324 is true. By monitoring the program flow the test data generator can determine if the intended path was taken. If not, the branch function corresponding to the place where the flow first deviated from the path is reconsidered. The input values are manipulated such that the branch function becomes smaller and when it becomes negative or zero, depending on the \( \text{relop} \), the intended branch is taken.

Korel’s branching functions have been addressed in a somewhat different manner in other work. For instance, in an article by Michael and McGraw, an objective function is used instead [29]. For the above code,
for instance, the objective function would correspond to:

\[ F(x) = \begin{cases} 
10 - a_{324} & \text{if } a_{324} < 10 \\
0 & \text{otherwise} 
\end{cases} \]

The objective function is constructed in such a way that when evaluated on a selected input it acts as a measure of how far the input is from satisfying the corresponding condition. When \( F(x) = 0 \) the condition is matched, otherwise, when \( F(x) > 0 \) it is not. The test data generator can use the objective function to guide the search for input values that are better. Input values when \( F(x) \) is closer to 0 are considered better.

Since branch constraints are considered only one at a time in Korel’s approach, it suffers from the fact that upon changing the flow at one point, the flow at an earlier point may also change. In other systems, such as ADTEST [30], this problem is handled by considering an objective function \( F(x) \) composed of the sum of objective functions for the branch constraints \( F_i(x) \), that is, \( F(x) = F_1(x) + F_2(x) + \cdots + F_n(x) \). Unfortunately, the complete objective function cannot be evaluated until the positions corresponding to the different \( F_i \)'s have been reached. Therefore, the different \( F_i \)'s are successively added as soon they are reached. For example, assume that when executing a program only branch constraints on line 3, 6, and 7 are reached. Therefore, \( F(x) = F_3(x) + F_6(x) + F_7(x) \).

### 2.1.3 Hybrid Approach

Common to many of the techniques based on actual execution is the lack of global scope, since the complete path constraint is not considered at once. Many techniques rely more on a heuristic approach to finding input traversing a certain path than actually trying to find a valid path constraint. However, a very interesting hybrid between symbolic and actual execution is presented by Gupta et al. [6]. Instead of deriving an exact symbolic representation of a path predicate an approximate one is derived. A benefit of this is that the complete path constraint is considered at once, as in the case of symbolic execution. Furthermore, complicated issues such as functions without source code are not a problem, since the approximate symbolic path predicate is derived upon program execution, as in actual execution.

Another example of a hybrid approach is dynamic domain reduction (DDR) [31]. It is a technique developed to overcome some of the limitations of constraint-based testing [32] (CBT) which was used in mutation testing system. CBT relies only on static information (such as symbolic execution
and control-flow graphs), while DDR also take dynamic information into account. In this way DDR can better handle arrays and pointer which often are used in a dynamic fashion. DDR have much similarities with finite domain reduction used in constraint programming [33].

Visvanathan and Gupta report a similar technique [34] to Zhang’s for handling programs with pointers. The technique performs two passes over the program. In the first pass the structure or shape of all pointer variables are generated. In the second pass the actual values of the structures pointed to are generated using the hybrid approach by Gupta et al.

2.1.4 Goal-oriented vs. Path-oriented

The main objective when generating test data is to increase coverage of the program with respect to the coverage criterion in use. Coverage is often correlated to the program’s control-flow graph or data-flow graph and therefore it is natural to generate test data in a path-oriented manner. For example, a set of paths in the control-flow graph is selected. Each path is then given to the test data generator, which constructs a predicate from the path according to the techniques described earlier.

A drawback with this approach is that paths of a program may be infeasible, meaning that they are non-executable and that there are no inputs that traverse the path. In turn, to achieve full coverage several so called path covers must be exploited before finding a set of feasible paths that fulfills the criterion.

Despite this, most of the test data generators reported in the literature seem to be path-oriented, probably because of the great advantage of the path actually representing an unrolled version of the program. This makes it much easier to determine the shape of array and pointer structures.

The alternative is called goal-oriented test data generator [35, 36]. Instead of requiring a specific path to be traversed, the goal is to traverse certain nodes in the flow graph. This reduces the risk of encountering infeasible paths, however, it complicates the determination of shape.

Gotlieb and Denmat [37] uses a modified version of single static assignment to handle pointer variables their suggested goal-oriented test generator.

2.1.5 Syntax-oriented

The literature reports on test data generators that uses coverage criteria which are not path oriented. For example, in compiler testing the grammar of the language supported by the compiler can be used to generate
2.2 Constraint Solver

The responsibility of a constraint solver is to find input values for the program such that a given path is traversed. Korel [28] suggests a gradient descent approach referred to as alternating variable. Here, each input variable is considered one at a time. Exploratory moves, that is, small moves, around the input variable are made in turn to predict if a larger step in some direction on this variable can improve the branch function’s value.

If the algorithm determines a direction to proceed, a larger step is taken. A series of large moves are made along the direction as long as the branch function is improved. In this manner, the algorithm cycles through the input variables in order to find a solution.

A problem, though, is that the search algorithm can get caught in a local minimum, not realizing that a solution is found elsewhere. On the positive side, this technique makes no assumptions about the composition of the branch function whatsoever.

Because of very large input domains and the chance of falling into a local minimum, it is common to have a limit on the number of iterations to perform before the search is abandoned. If a solution is not found within the limit it is considered unknown whether the constraint is solvable or not.

Deason et al. [40] presents a rule-based approach to constraint solving. The solver iteratively executes the program under test. In each iteration it checks whether a condition in the program matches any of the rules of the solver. Each rule is associated with a certain action to take, for example, incrementing or decrementing a specific input variable. The technique is local in that it does not consider the whole path constraint at once.

Another technique suggested for constraint solvers in TDG is a general-purpose optimization method called simulated annealing [41]. Starting from some initial input the algorithm selects solution candidates which lie in the neighborhood. Solutions improving the objective function are always accepted, however, solutions that are not leading to an improve-
ment are accepted in a controlled manner. The basic idea is that in spite of accepting a candidate with an unfavorable change in the objective function the candidate may improve the search in the long run. In accepting an inferior solution, the algorithm might escape from a local optimum.

A parameter, known as the temperature, controls how inferior solutions are accepted. Initially, when the temperature is high, the algorithm accepts, almost unexclusively, all candidates. As the temperature gradually cools off the acceptance of inferior candidates is constrained. When the process eventually freezes, no inferior solutions are accepted; the search is reduced to simple hill climbing.

Similarly, approaches using genetic algorithms also focus on improving the ability of escaping a local optimum. Here the solver starts out with some initial (random) inputs, referred to as a population. The population is exposed to evolution such that better inputs have a higher chance of having offspring. In this way, genetic algorithms perform an evolutionary search through the input domain. An objective function evaluates how close to a solution the generated inputs are. The closer a solution, the greater the likelihood that it will lead to a solution, therefore such inputs have a higher probability of being genetically combined to produce new input.

Michael et al. [29] report on interesting results when comparing test data generation based on genetic algorithms with random generators and gradient-descent generators. It was found that non-random techniques performed much better, in terms of coverage, on complex programs than random testing. Genetic generators were slightly better than the gradient-descent generator used. This is probably because genetic algorithms make no presumptions regarding the actual program.

Another strategy reported by Gotlieb and Denmat is to represent the path constraint as a constraint logic program (CLP). Constraint programming [33] is a collection of techniques that can be used to solve constraints of various forms. Gotlieb and Denmat represented the constraints using so-called finite domain constraints, meaning their technique is limited to linear integer arithmetics (decimal numbers can be scaled to integers).

A CLP-solver enumerates the problem variables in intelligent ways such that the problem can be more and more constrained. The fact that CLP-solvers work in this integral way makes them especially suitable for solving so-called integer problems. Traditionally, approaches such as the Simplex method in combination with branch & bound [42,43] have been used solve (mixed) integer problems. The constraint solver reported by Zhang and Wang [27] is based on a mixed-integer solver called lp_solve [44].

The hybrid approach for finding path constraints suggested by Gupta et
2.2 Constraint Solver

al (see section 2.1.3), was initially based on a constraint solver with limited capabilities. In later publications a new constraint solving method called the universal numerical approach (UNA) has been developed [7, 8]. As previously described, the initial step is to approximate the path constraint by a conjunction of linear constraints. The advantage is that when the true path constraint is linear, the solution can definitely be found. They justify the linearization by results showing that a large portion of analyzed Fortran code is in fact composed of linear constraints. That is, it is not often that two or more variables are multiplied or in some other way form non-linear constraints. After the linearization, the constraints are given to UNA which returns a solution. If the approximate solution is not a solution to the original problem, a relinearization is performed in a similar manner to the famous Newton-Raphson algorithm for finding roots.

The UNA itself is an iterative refinement approach based on the least square problem. A least square solution is sought for the approximated constraints. By iteratively moving violating constraints along their normal vector the least square solution is forced into the solution space for the initial linear problem.

Unfortunately, as will be shown in successive chapters of part I, there are circumstances where UNA fails to find a solution to a set of constraints. Therefore, an alternative to UNA based on the Simplex method is also explored.
Chapter 3
Introduction to the Unified Numerical Approach

— Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

Albert Einstein

In an article by Gupta et al. [6] a test data generator based on a hybrid technique between symbolic and actual execution is presented. They propose that a linear representation of each branch constraint is derived to form a linear equation system representing the complete path constraint. This system is then solved using Gaussian elimination [45], and therefore, limited to real variables. Often, however, this technique suffered from the fact that the derived equation system was over- or under-determined. To overcome the limitations, Gupta et al. developed a new constraint-solving technique known as the Unified Numerical Approach (UNA) [7, 8]. In this theses it is shown that UNA has the following weaknesses:

(a) There are circumstances where UNA fails to find a solution to a mixed integer problem, given that there is at least one.

(b) The number of iterations used by UNA to find a real value solution has a worst case bounded by infinity for all problem sizes.

My contributions to the TDG problem are

1. the weaknesses of UNA are analyzed and shown through formal mathematical proofs and a number of examples run in a prototype implementation.
2. it is suggested that a constraint solver based on the a linear program-
ing technique, for instance the simplex method, in combi-
nation with branch-and-bound and/or a cutting-plane method, is
used instead of UNA.

The performance and reliability of the TDG proposed by Gupta et al.
will improve if instead of UNA the proposed constraint solver is used
since

(a’) it guarantees finding a solution in any circumstance, as long as there
is one.

(b’) it will improve the worst case upper bound on the iteration count.

The rest of the work in part I is organized as follows. Section 3.1 covers
the basic concepts and terminology used in this part. In section 3.2 an
introduction and background to UNA is given, and it is formally defined in
section 3.3. In chapter 4 we analyze the complexity and incompleteness
of UNA. Here we also present our alternative algorithm and a comparison
between the two. Finally, in chapter 6 conclusions and future work are
presented.

3.1 Basic Concepts and Notations

Vectors are denoted with lowercase Latin letters, e.g. $x$, $r$, and $d$. The
elements of a vector are referred to by subscripting, as in $x_i$, while a
parenthesized superscript notation denotes a variable at a specific itera-
tion, e.g. $x^{(i)}$ refers to vector $x$ at iteration $i$.

Matrices are written with uppercase Latin letters, e.g. $A$. The notation
$a_{i \cdot}$ references the $i$:th row of $A$, while $a_{\cdot j}$ means the $j$:th column. A matrix
element at row $i$ and column $j$ in matrix $A$ is denoted by $a_{ij}$.

Here the problem of test data generation has the following form,

Find $x = (x_1, \ldots, x_n)^T$
subject to $Ax \geq b$

$x_i \in \mathbb{Z}, i \in \mathcal{I}$
$x_i \in \mathbb{R}, i \notin \mathcal{I}$,

(3.1)

where $A$ is an $m \times n$ matrix, $m > n$, $b$ is a vector of length $m$, and
$\mathcal{I} \subseteq \{1, \ldots, n\}$ is an set of indices pointing out the variables that are con-
strained to integers ($\mathbb{Z}$). This problem definition is a slight simplification
over the one used in the UNA articles [7,8], as far as the relational operator
is concerned. The original description is more general and allows =, ≤, <, and > as well. This, however, will not affect the results presented here, since both ≤ and = easily can be converted to ≥. Furthermore, just a minor modification to the algorithm is needed in order to handle > (or <), and again, this change will not have any impact on our results. Thus, having the problems on the form (3.1) will merely simplify our discussion.

3.1.1 Equation Systems

Any system of linear inequalities, such as that shown in (3.2), can be transformed to a system of equalities by introducing slack variables (3.3).

\[
\begin{align*}
-x_1 + x_2 & \geq 0 \\
x_1 + x_2 & \geq 0 \\
x_2 & \geq -2
\end{align*}
\]  
(3.2)

\[
\begin{align*}
-x_1 + x_2 - s_1 &= 0 \\
x_1 + x_2 - s_2 &= 0 \\
x_2 - s_3 &= -2
\end{align*}
\]  
(3.3)

\[s_1, s_2, s_3 \geq 0\]

The slack variables are interpreted as the amount that has to be subtracted from the left hand side in order to make it equal to the right hand side. A more convenient form of writing (3.3) is to put it in matrix notation. We have

\[
\begin{pmatrix}
-1 & 1 & -1 & 0 & 0 \\
1 & 1 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 & -1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
s_1 \\
s_2 \\
s_3
\end{pmatrix}
= 
\begin{pmatrix}
0 \\
0 \\
-2
\end{pmatrix}.
\]  
(3.4)

Let \(A\), \(S\), \(x\), and \(s\) be defined as follows.

\[
A = \begin{pmatrix}
-1 & 1 \\
1 & 1 \\
0 & 1
\end{pmatrix}, \quad S = \begin{pmatrix}
-1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & -1
\end{pmatrix},
\]

\[
x = \begin{pmatrix}
x_1 \\
x_2
\end{pmatrix}, \quad s = \begin{pmatrix}
s_1 \\
s_2 \\
s_3
\end{pmatrix}.
\]
Chapter 3 Introduction to the Unified Numerical Approach

Then by writing \((A|S)\) and \((x|s)\) we mean the horizontal and the vertical concatenation. That is,

\[
(A|S) = \begin{pmatrix} -1 & 1 & -1 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \end{pmatrix}, \quad (x|s) = \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}.
\]

(3.5)

3.2 Introduction to UNA

The Unified Numerical Approach (UNA) is a method developed by Gupta et al. [7, 8] for finding real solutions, as well as mixed integer solutions, to an over-determined system of linear equalities and inequalities. The goal of the method is to create a sequence of points \(x^{(0)}, x^{(1)}, \ldots, x^{(k)}\), such that each point successively comes closer to a solution. Finally, the point \(x^{(k)}\) will be a feasible solution. In the first iteration the least square solution [45] of the system is calculated. If this is not a feasible solution, a new system is formed by systematically moving the linear constraints violating the current solution. The new system is then used in the next iteration.

Before looking into the details we will give two examples of how to solve equation system (3.2). The first example shows a solution in an ad hoc manner, while the second applies the UNA algorithm. The latter example can also be found in [7,8].

Example 3.1. Writing the slack version of (3.2) in matrix notation \(Ax' = b\) we have

\[
A = \begin{pmatrix} -1 & 1 & -1 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix},
\]

\[
x' = (x_1 \ x_2 \ s_1 \ s_2 \ s_3)^T
\]

A straightforward approach in solving this system is to consider it as a system of three unknown variables, referred to as basic variables, and have the remaining as free variables. From linear algebra [45] we know that the columns corresponding to the basic variables must be linearly independent. Consequently, if \(a_{k1}, a_{k2}, \) and \(a_{k3}\) are three linearly independent columns we have the following system in terms of \(x'_{k4}\) and
3.2 Introduction to UNA

\[ x'_{k_5} \cdot \]

\[(a_{k_1} \ a_{k_2} \ a_{k_3}) \begin{pmatrix} x'_{k_1} \\ x'_{k_2} \\ x'_{k_3} \end{pmatrix} = b - (a_{k_4} \ a_{k_5}) \begin{pmatrix} x'_{k_4} \\ x'_{k_5} \end{pmatrix} \]  \hspace{1cm} (3.6)

For example, if we select the columns corresponding to \(x_1\), \(x_2\), and \(s_1\), we get

\[
\begin{pmatrix}
-1 & 1 & -1 \\
1 & 1 & 0 \\
0 & 1 & 0 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
s_1 \\
\end{pmatrix}
= 
\begin{pmatrix}
0 \\
0 \\
-2 \\
\end{pmatrix}
- 
\begin{pmatrix}
0 & 0 \\
-1 & 0 \\
0 & -1 \\
\end{pmatrix}
\begin{pmatrix}
s_2 \\
s_3 \\
\end{pmatrix}
\]  \hspace{1cm} (3.7)

which gives us the solution

\[
\begin{pmatrix}
x_1 \\
x_2 \\
s_1 \\
\end{pmatrix}
= 
\begin{pmatrix}
2 + s_2 - s_3 \\
-2 + s_3 \\
-4 - s_2 + 2s_3 \\
\end{pmatrix},
\]  \hspace{1cm} (3.8)

where \(s_2\) and \(s_3\) can have any value as long as the slack constraint holds \((s_i \geq 0)\). Letting \(s_2 = 0\) and \(s_3 = 2\) gives us the feasible solution \((x_1, x_2, s_1)^T = (0, 0, 0)^T\).

Depending on the size and the form of the system, the approach used in example 3.1 can be quite complex. When \(A\) has \(m\) rows and \(n\) columns, there are \(\frac{n!}{m!(n-m)!}\) combinations of basic variables. For this example there are 10 combinations, but letting \(m = 5\) and \(n = 9\) we will have \(\frac{9!}{5!4!} = 126\). Another difficulty can be to assign values to the free variables, so that they do not violate the slack constraint. In this example this was not a problem, but when some variables have integer constraints, i.e. they can only take on integer values, this becomes cumbersome. UNA handles the slack variable problem by systematically incrementing the slack until a solution is found. In the following example, inspired by [7, 8] we will see how the method works.

**Example 3.2.** Assume we have system (3.2) in matrix form \(Ax \geq b\), where

\[
A = \begin{pmatrix}
-1 & 1 \\
1 & 1 \\
0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\end{pmatrix}
= 
\begin{pmatrix}
0 \\
0 \\
-2 \\
\end{pmatrix}.
\]  \hspace{1cm} (3.9)

By introducing slack variables we have \(Ax = b + s, \ s \geq 0\). Our intention is to find a real solution to the system by identifying the least square solution of \(Ax = b + s\), starting with \(s = 0\). By systematically changing \(s\) we will see how we can find a feasible solution, i.e. a point within the shaded region of figure 3.1.
Chapter 3 Introduction to the Unified Numerical Approach

Figure 3.1: For each iteration the least square solution comes closer to the feasible region.

Figure 3.2: The violating constraints are moved along its normal vectors dragging the solution towards the feasible region.
3.2 Introduction to UNA

Table 3.1: The least square solutions and their corresponding residuals.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(r_1)</th>
<th>(r_2)</th>
<th>(r_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-0.6667</td>
<td>-0.6667</td>
<td>-0.6667</td>
<td>1.3333</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-0.2222</td>
<td>-0.2222</td>
<td>-0.2222</td>
<td>1.7778</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>-0.0741</td>
<td>-0.0741</td>
<td>-0.0741</td>
<td>1.9259</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-0.0247</td>
<td>-0.0247</td>
<td>-0.0247</td>
<td>1.9753</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>-0.0082</td>
<td>-0.0082</td>
<td>-0.0082</td>
<td>1.9918</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-0.0027</td>
<td>-0.0027</td>
<td>-0.0027</td>
<td>1.9973</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>-0.0009</td>
<td>-0.0009</td>
<td>-0.0009</td>
<td>1.9991</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>-0.0003</td>
<td>-0.0003</td>
<td>-0.0003</td>
<td>1.9997</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>-0.0001</td>
<td>-0.0001</td>
<td>-0.0001</td>
<td>1.9999</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>-0.0001</td>
<td>-0.0001</td>
<td>-0.0001</td>
<td>1.9999</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Increasing the slack element \(s_i\) for a constraint is the same as moving the constraint corresponding to \(s_i\) along its normal vector (figure 3.2).

In the first iteration the slack vector \(s^{(0)} = 0\). Therefore, we simply find the least square solution of \(Ax^{(0)} = b\), i.e. \(x^{(0)} = (A^TA)^{-1}A^Tb = \left(\begin{array}{c} 0 \\ -2/3 \end{array}\right)\). In figure 3.1 we can easily see that the solution is infeasible.

The residual \(r = Ax - b\) is a measure of how far a point \(x\) is from a feasible solution. It is easy to prove that \(x\) is feasible if and only if \(r \geq 0\). In our example we have \(r^{(0)} = (-2/3, -2/3, 4/3)^T < 0\), and thus \(x^{(0)}\) is infeasible.

A negative value for a component \(r_i\) means that the corresponding equation \(a_i x \geq b_i\) is not satisfied. Our objective for the next iteration is to force \(r \geq 0\). We do this by incrementing the components of \(s\) corresponding to the negative components of \(r\), with an increment of \(|r_i|\). This gives us

\[
s^{(1)} = \begin{pmatrix} 2/3 \\ 2/3 \\ 0 \end{pmatrix}.
\]

Again, we find the solution of \(Ax^{(1)} = b + s^{(1)}\). We continue in this manner until we have \(r \geq 0\). In table 3.1 we see how \(x\) and \(r\) change in each iteration, and in the eleventh iteration we stop with \(r \geq 0\) and \(x = \left(\begin{array}{c} 0 \\ 0 \end{array}\right)\).


Chapter 3 Introduction to the Unified Numerical Approach

3.3 UNA Algorithm

Gupta et al. present a very brief description of UNA. The description is offered only through an example, and thus there are many aspects of the UNA which are not discussed in their publications [7, 8], for instance, how to handle linearly dependent columns in \( A \). Therefore, we assume that matrix \( A \) has full column rank, that is, it consists only of linearly independent columns. Below we give our interpretation of the UNA with the following formal description.

**Algorithm 3.1.**

Find \( x = (x_1, \ldots, x_n)^T \)

subject to

\[
Ax \geq b \\
x_i \in \mathbb{Z}, i \in \mathcal{I} \\
x_i \in \mathbb{R}, i \notin \mathcal{I},
\]

where \( A \) is an \( m \times n \) matrix with full column rank, \( m > n \), \( b \) is a vector of length \( m \), and \( \mathcal{I} \) is an index set for the integer constraints.

1. Let the iteration counter \( k = 0 \), and let the initial slack vector \( s^{(0)} = 0 \).
2. Find the non-rounded solution \( \hat{x}^{(k)} \), defined by the least square solution of \( A\hat{x}^{(k)} = b + s^{(k)} \).
3. Find the rounded solution \( x^{(k)} = (x_1, \ldots, x_n) \), where

\[
x_i = \begin{cases} 
\lfloor \hat{x}_i + 0.5 \rfloor & i \in \mathcal{I} \\
\hat{x}_i & \text{otherwise}
\end{cases}
\]

4. Calculate the residual \( r^{(k)} = Ax^{(k)} - b \).
5. if \( r^{(k)} \geq 0 \) terminate with \( x^{(k)} \) as solution.
6. Let the direction vector \( d^{(k)} = (d_1, \ldots, d_n) \), where

\[
d_i = \begin{cases} 
|r_i| & \text{if } r_i < 0 \\
0 & \text{otherwise}
\end{cases}
\]

7. Let \( s^{(k+1)} = s^{(k)} + d^{(k)} \)
8. Let \( k := k + 1 \)
9. Go to step 2.
3.3 UNA Algorithm

We refer to $\hat{x}^{(i)}$ as an internal point or a non-rounded solution. $x^{(i)}$ is called an external point or a rounded solution. Internal points and external points are equivalent when $x_i \in \mathbb{R}$.

3.3.1 Implementation

We have implemented the above algorithmic description of UNA in Matlab\textsuperscript{1}. During implementation, when comparing our results with those produced by Gupta et al. in their article on UNA [7], we realized that the algorithm is highly dependent on the floating point precision, due to a very low convergence. In example 3.2 above we see that a solution is found after 11 iterations, but in fact, if full (Matlab) floating point precision is used it will take 36 iterations. In order to make our algorithm produce the same number of iterations that Gupta shows, the direction vector needs to be rounded to 4 decimals. Therefore the following line was added to the Matlab code right after step 6 of the algorithm.

\[
d = \text{round}(d*10000)/10000;
\]

All results presented here come from our prototype implementation.

\textsuperscript{1}A two-dimensional version of UNA has also been implemented in MetaPost in order to produce the geometrically exact figures used in this document.
Chapter 4

Complexity and Incompleteness of UNA

— When you give for an academic audience a lecture that is crystal clear from alpha to omega, your audience feels cheated and leaves the lecture hall commenting to each other: 'That was rather trivial, wasn’t it?’ The sore truth is that complexity sells better.

Edsger Dijkstra

Two important algorithmic properties of the UNA are analyzed in this chapter, namely complexity and completeness. Complexity refers to the time it takes to find a solution. Specifically, we are interested in the number of iterations performed before a solution is determined, as well as the complexity of each iteration. Completeness refers to whether the algorithm definitely finds a solution when there is one.

4.1 Complexity of UNA

In this section we will examine the upper bound of the number of iterations needed by UNA in order to find a solution. For simplicity, we will consider the case when \( x \in \mathbb{R}^n \), i.e. a real solution is sought.

Assume that we have the system shown below (4.1). We have chosen \( A \) and \( b \) to be two-dimensional vectors to make it easy to draw them in a graph. Figure 4.1 illustrates the result after the first iteration. In order for \( x \) to be a feasible solution, \( r \) must be greater than or equal to 0. Thus, \( r \) must lie inside the shaded region.

\[
Ax = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} x \geq \begin{pmatrix} 2 \\ 4 \end{pmatrix} = b
\]

(4.1)
Figure 4.1: The residual in the first iteration is always perpendicular to $A$.

Clearly, this is not the case, and therefore, the next step is to identify the direction vector, which will give us our next slack. The direction vector is $d = \begin{vmatrix} 0 \\ r_2 \end{vmatrix}$, since the only negative component of $r$ is $r_2$. This situation is reflected in figure 4.2. The vector $d$ is used to increment the slack vector, and thus move $b + s$ upwards as shown in figure 4.3. The next solution $x^{(1)}$ is derived using $b + s^{(1)}$.

Running the system through our implementation of UNA, it gives us the solution $x = 4$ after 30 iterations. In figure 4.4, which illustrates the three first iterations, we see that the direction vector becomes shorter for each iteration. This is why the step size between each $Ax^{(i)}$ gets smaller.

With some fairly simple geometrical reasoning we can see that the angle between $A$ and the two axes is directly coupled to the step size. In figure 4.4 we have marked the angle between $A$ and the $e_1$ axis with $\alpha$.

In the system given below we have modified $A$ such that $\alpha$ is smaller in comparison to (4.1).

$$Ax = \begin{pmatrix} \frac{5}{2} \\ 1 \end{pmatrix} x \geq \begin{pmatrix} 2 \\ 4 \end{pmatrix} = b, \quad (4.2)$$

Figure 4.5 shows the first 14 iterations of the new system, and not until iteration 1118 do we find the solution $x = 8$ (rounded). If we let $\alpha$ go to 0 the number of iterations goes to infinity. In fact, $\alpha = 0$ corresponds to $A = \begin{pmatrix} a_1 \\ 0 \end{pmatrix}$, where $a_1 \neq 0$, which means that in the system $\begin{pmatrix} a_1 \\ 0 \end{pmatrix} x \geq \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$, we have $0x \geq b_2$, which only holds if $b_2 = 0$. If $b_2 \neq 0$ the equation is
Figure 4.2: The direction vector points to the closest point where the residual is greater than the zero vector.

Figure 4.3: The next point $x^{(1)}$ is derived by using $b + s^{(1)}$. 
Figure 4.4: The three first iterations of system (4.1). It can be seen that the distance between each point is smaller.

Figure 4.5: With a smaller angle the number of iterations grows. Here are the first 14 iterations of system (4.2).
4.2 Incompleteness of UNA

inconsistent, thus explaining why the number of iterations goes to infinity. In figure 4.4, we see that the angle between \( r^{(0)} \) and \( d^{(0)} \) is also \( \alpha \). Measuring this angle reflects the general case when \( A \) is an \( m \times n \) matrix, \( m > n \). We see that \( \alpha = 0 \) means that \( r \) and \( d \) are aligned, which means that the step size will be 0, which in turn means that the equation system is inconsistent. It is simple to extend (4.2) to a larger system that will have the same properties, thus showing that the UNA does not have an upper bound on the number of iterations.

In each operation of the UNA the least square solution is calculated, which can be performed in the following way.

\[
\hat{x}^{(i)} = (A^T A)^{-1} A^T (b + s^{(i)})
\]

The indicated expression \( H \) can be calculated with a complexity of \( O(n^3) \) additions and multiplications. Most of the work comes from finding the inverse of \( A^T A \). However, since the matrix \( A \) does not change during the search, \( H \) can be computed in advance and therefore the only expression which affects the iteration complexity is \( H(b + s^{(i)}) \), which can be calculated with \( O(mn) \) additions and multiplications.

4.2 Incompleteness of UNA

We have seen that the complexity of UNA depends on how input is composed, and that UNA performs an infinite number of iterations in the worst case — regardless of problem size. Another important question, left unanswered in \([7, 8]\), is whether UNA is complete. That is, does it always find a solution when there is one? Unfortunately, the answer is no. First, we give an informal explanation of the incompleteness, before we provide a formal proof.

UNA will continue generating points until a feasible solution is found. Thus, if we can make UNA loop forever, even though there is a solution, we have shown the algorithm is incomplete. Assume that UNA has generated the following sequence of points \( P = \langle p_0, p_1, \ldots, p_j, p_{j+1}, \ldots, p_k \rangle \). Consider now what will happen if \( p_j = p_k \). Because each point \( p_i \) is generated systematically, based on the residual of the previous point \( p_{i-1} \), we intuitively expect that whenever \( p_j = p_k \) we will also have \( p_{j+1} = p_{k+1} \). That is, if we ever revisit a point \( p_j \), then the next point must be \( p_{j+1} \). Consequently, \( P \) will end in an infinite repeating subsequence \( \langle p_j, p_{j+1}, \ldots, p_{k-1}, p_j, \ldots \rangle \). In fact, we will show that if \( p_k \) is equal to any other visited point in \( P \), besides the first \( p_0 \), then UNA will never
terminate. This, together with a counter example will show that UNA is incomplete.

Before we can work out a full proof, we need to understand some of the properties concerning the direction- and the slack vectors. In observation 4.1 and 4.2 below we have a special interpretation of the operators > and ≥ on vectors. When saying \( a \geq b \) we mean that \( \forall i : a_i \geq b_i \). That is, all components of vector \( a \) must be greater than or equal to its corresponding component in vector \( b \). However, when \( a > b \) we only require that there exists at least one component in \( a \) that is greater than its correspondent. In other words, when \( a > b \) we mean \( \forall i : a_i \geq b_i \land \exists j : a_j > b_j \).

**Observation 4.1.** For a point \( x^{(i)} \) the following is true.

(i) \( d^{(i)} \geq 0 \).

(ii) \( d^{(i)} = 0 \) iff \( x^{(i)} \) is feasible.

*Proof.* It follows from the definition of \( d^{(i)} \).

**Observation 4.2.** For some \( k > 0 \), and \( d^{(k)} > 0 \), let \( x^{(0)}, x^{(1)}, \ldots, x^{(k)} \), be a sequence of points generated by UNA, then

\[ s^{(k+1)} > s^{(k)} > 0. \]

*Proof.* By definition of \( s^{(k)} \) and observation 4.1 we have

\[ s^{(k+1)} = s^{(k)} + d^{(k)} > s^{(k)} > 0. \]

**Observation 4.3.** If UNA has generated the following sequence of internal points

\[ \hat{x}^{(0)}, \hat{x}^{(1)}, \ldots, \hat{x}^{(j)}, \hat{x}^{(j+1)}, \ldots, \hat{x}^{(k)} \]

where \( d^{(k)} > 0 \), and \( \hat{x}^{(k)} = \hat{x}^{(j)} \), \( k > j \geq 0 \), then the next generated point \( \hat{x}^{(k+1)} = \hat{x}^{(j+1)} \).

*Proof.* By definition of least square solution [45] we have

\[ \hat{x}^{(i)} = (A^T A)^{-1} A^T (b + s^{(i)}), \forall i \]
4.2 Incompleteness of UNA

Since \( \hat{x}^{(k)} = \hat{x}^{(j)} \) we have

\[
(A^T A)^{-1} A^T (b + s^{(k)}) = (A^T A)^{-1} A^T (b + s^{(j)})
\]
\[
A^T (b + s^{(k)}) = A^T (b + s^{(j)})
\]
\[
A^T (s^{(k)} - s^{(j)}) = A^T (b - b)
\]
\[
A^T (s^{(k)} - s^{(j)}) = 0
\]

From observation 4.2 we know \( s^{(k)} = s^{(j)} + v \), \( v > 0 \). This gives us

\[
A^T (s^{(j)} + v - s^{(j)}) = 0
\]
\[
A^T v = 0,
\]

which means that \( v \) is orthogonal to the columns of \( A \). In turn, this means that changing the slack by \( v \), will not improve our solution, i.e. \( \hat{x}^{(j)} = \hat{x}^{(k)} \). Moreover,

\[
\hat{x}^{(k)} = \hat{x}^{(j)} \implies r^{(k)} = r^{(j)} \implies d^{(k)} = d^{(j)},
\]

This means that, for two equal points we will have the same residual and direction vector. Knowing that, we get

\[
s^{(k+1)} = s^{(k)} + d^{(k)}
\]
\[
= s^{(k)} + d^{(j)}
\]
\[
= s^{(j)} + v + d^{(j)}
\]
\[
= s^{(j+1)} + v
\]

which implies

\[
\hat{x}^{(k+1)} = (A^T A)^{-1} A^T (b + s^{(k+1)})
\]
\[
= (A^T A)^{-1} A^T (b + s^{(j+1)} + v)
\]
\[
= (A^T A)^{-1} A^T (b + s^{(j+1)}), \quad \text{since } A^T v = 0
\]
\[
= \hat{x}^{(j+1)}
\]

To illustrate the the above observation, assume that we have generated a sequence of points with the following labels:

\[
1, 2, 3, 4, 5, 3
\]
According to observation 4.3 the next generated point is 4. A consequence of this is that UNA will not terminate, but continue generating the infinite sequence of:

\[1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, \ldots\]

In the following observation we formally state this consequence.

**Observation 4.4.** If UNA has generated the following sequence of internal points

\[\hat{x}^{(0)}, \hat{x}^{(1)}, \ldots, \hat{x}^{(j)}, \hat{x}^{(j+1)}, \ldots, \hat{x}^{(k)}, \ldots\]

where \(\hat{x}^{(k)} = \hat{x}^{(j)}, k > j \geq 0\), then the algorithm will never find a feasible solution.

**Proof.** It follows by induction from observation 4.3.

From observation 4.4 we see that if UNA visits an internal point more than one time (two times if it is the starting point), then a feasible solution cannot be found, since the algorithm will never terminate. To prove that UNA is incomplete we only have to identify a system that behaves in this manner.

**Observation 4.5.** UNA does not always find a solution to a mixed integer problem. Thus, UNA is not complete.
4.2 Incompleteness of UNA

Proof. The following system \( Ax \geq b \), where \( x \in \mathbb{Z} \), and

\[
A = \begin{pmatrix}
\frac{1}{2} & -1 \\
-\frac{1}{2} & 1 \\
1 & 0
\end{pmatrix}, \quad \text{and} \quad b = \begin{pmatrix}
-\frac{3}{4} \\
\frac{1}{4} \\
0
\end{pmatrix},
\]

(4.3)
pictured in figure 4.6, has an infinite number of integer solutions \( x_1 = 2x_2 - 1 \), where \( x_2 \geq 1 \). The UNA algorithm, however, will generate the following sequence of external points:

\[
\begin{pmatrix}
0 \\
1
\end{pmatrix}, \begin{pmatrix}
0 \\
0
\end{pmatrix}, \begin{pmatrix}
0 \\
1
\end{pmatrix}, \begin{pmatrix}
0 \\
1
\end{pmatrix}, \cdots
\]

which corresponds to the internal points of:

\[
\begin{pmatrix}
0 \\
\frac{1}{2}
\end{pmatrix}, \begin{pmatrix}
0 \\
\frac{3}{8}
\end{pmatrix}, \begin{pmatrix}
0 \\
\frac{1}{2}
\end{pmatrix}, \begin{pmatrix}
0 \\
\frac{3}{8}
\end{pmatrix}, \begin{pmatrix}
0 \\
\frac{1}{2}
\end{pmatrix}, \cdots
\]

By observation 4.4 a feasible solution cannot be found, since UNA will end up in a loop just visiting the two internal points \( (0, \frac{1}{2}) \) and \( (0, \frac{3}{8}) \). As a consequence, UNA is also incapable of finding mixed integer solutions. To show this we add a third equation \( x_3 \geq b_4, x_3 \in \mathbb{R} \) to the above counter example.

\[
A = \begin{pmatrix}
\frac{1}{2} & -1 & 0 \\
-\frac{1}{2} & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{pmatrix}, \quad \text{and} \quad b = \begin{pmatrix}
-\frac{3}{4} \\
\frac{1}{4} \\
0 \\
b_4
\end{pmatrix},
\]

which will generate the following sequence of internal points:

\[
\begin{pmatrix}
0 \\
b_4
\end{pmatrix}, \begin{pmatrix}
0 \\
b_4
\end{pmatrix}, \begin{pmatrix}
0 \\
b_4
\end{pmatrix}, \begin{pmatrix}
0 \\
b_4
\end{pmatrix}, \begin{pmatrix}
0 \\
b_4
\end{pmatrix}, \cdots
\]

\[\square\]

4.2.1 Why Incomplete – Intuitive Explanation

The next point in every iteration of UNA, that is, the next least square solution, is given by the current direction vector \( d \). For two points \( p \) and
q, where \( p = q \) we will by definition have the same direction vector, that is \( \text{dir}(p) = \text{dir}(q) \).

Since the direction vector uniquely points out the next generated point, we can conclude that if UNA ever revisits a point then UNA will loop.

In fact, how \( d \) is calculated is not of interest. The reason that UNA loops is the systematic calculation of the direction vector. That is, every time a point \( p \) is visited we will have the same value \( d \).

4.2.2 Why Incomplete – Mathematical Explanation

Intuitively we can view the least square error solution of \( Ax = b \), where \( x \in \mathbb{R} \), as a projection of \( b \) onto \( A \). This means that incrementing \( b \) with a direction vector that is orthogonal to the columns of \( A \), \( (A^t d = 0) \), will not change the least square solution. By definition of the least square solution we have:

\[
(A^t A)^{-1} A^t (b + d) = (A^t A)^{-1} A^t b, \quad \text{since} \quad A^t d = 0
\]

This means that any increment that is parallel to the residual (or orthogonal to \( A \)) is in vain. Figure 4.7 illustrates the situation. The direction vector can be expressed as in a base of the residual and \( A \), in this case it results in \( d^{(i)} = d^{(i)}_r + d^{(i)}_A \). The projection of \( d^{(i)} \) onto \( A \) must therefore be \( A x^{(i+1)} = A x^{(i)} + d^{(i)}_A \).

Thus, instead of adding \( d^{(i)} \) to the slack, we could equally well add \( d^{(i)}_A \). The difference is that instead of \( b + s \) stepping along the vertical side of the shaded area, it will step along the horizontal side.

Thus, if the points in iteration \( j \) and \( k \) are equal, then the difference in their slack variables \( v = s^{(k)} - s^{(j)} \) must be orthogonal to the columns of \( A \).

\[
(A^t A)^{-1} A^t (b + s^{(k)}) = (A^t A)^{-1} A^t (b + s^{(j)})
\]
\[
A^t (b + s^{(k)}) = A^t (b + s^{(j)})
\]
\[
A^t (s^{(k)} - s^{(j)}) = A^t (b - b)
\]
\[
A^t (s^{(k)} - s^{(j)}) = 0
\]

By definition of the least square solution we know that \( A \neq 0 \). Further, by definition of the UNA we know that \( s^{(k)} - s^{(j)} \neq 0 \). Altogether, this means that \( v \) is orthogonal to the columns of \( A \) or to put it in an other way \( v \) is a multiple of the residual to the least square solution. In turn,
4.2 Incompleteness of UNA

Figure 4.7: Components of the direction vector that is orthogonal to $Ax$ does not contribute to the solution.
this means that all increments added between the two points have been in vain.

Since the two points of iteration \( j \) and \( k \) are equal they also give the same direction vector:

\[
x^{(j)} = x^{(k)} \implies d^{(j)} = d^{(k)}
\]

Therefore, the next generated point is \( x^{(k+1)} = x^{(j+1)} \).

Until this point we have considered only real numbered solutions for \( Ax \geq b \), i.e. \( x_i \in \mathbb{R} \). Now, will our reasoning still hold if we constrain some \( x_i \):s to \( \mathbb{Z} \)? Yes, it will! Let us, as in the UNA algorithm, call the non-rounded least square solution for \( \hat{x} \) and the rounded solution for \( x \), then for two points \( \hat{x}^{(j)} = \hat{x}^{(k)} \) we have:

\[
\hat{x}^{(j)} = \hat{x}^{(k)} \implies x^{(j)} = x^{(k)} \implies d^{(j)} = d^{(k)}
\]

As in the real numbered case the two slack variables for the two least square solutions are composed such that \( \hat{x}^{(j)} = \hat{x}^{(k)} \). Therefore, an increment with the same direction vector gives us

\[
\hat{x}^{(j+1)} = \hat{x}^{(k+1)}
\]

### 4.2.3 Examples

Table 4.1 shows all six UNA iterations of a simple system that does not loop. It should be noted that whether a system is looping cannot be decided from the external solution, that is, the converse of \( \hat{x}^{(j)} = \hat{x}^{(k)} \implies x^{(j)} = x^{(k)} \) is not always true. The same external solution \( \frac{2}{0} \) is visited twice. However, the looping condition is determined from the internal solutions, which are all unique.

Table 4.2 lists the first five UNA iterations of system (4.3). After four iterations we can conclude that this system will loop, since \( \hat{x}^{(1)} = \hat{x}^{(3)} \).
4.2 Incompleteness of UNA

Table 4.1: The result of running UNA simple system that does not loop.

\[ A = \begin{pmatrix} -4 & -4 \\ 2 & 3 \end{pmatrix}, \text{ and } b = \begin{pmatrix} -11.2 \\ 4.8 \end{pmatrix}, \]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(s^{(i)})</th>
<th>(\hat{x}^{(i)})</th>
<th>(x^{(i)})</th>
<th>(r^{(i)})</th>
<th>(d^{(i)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\begin{pmatrix} 0 \ 0 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3.1111 \ 0.3556 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} -0.8 \ 1.2 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0.8 \ 0 \end{pmatrix})</td>
</tr>
<tr>
<td>1</td>
<td>(\begin{pmatrix} 0.8 \ 0 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2.7556 \ -0.1778 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} -0.8 \ 1.2 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0.8 \ 0 \end{pmatrix})</td>
</tr>
<tr>
<td>2</td>
<td>(\begin{pmatrix} 1.6 \ 0 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2.4000 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3.2 \ -0.8 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0 \ 0 \end{pmatrix})</td>
</tr>
<tr>
<td>3</td>
<td>(\begin{pmatrix} 1.6 \ 0.8 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2.0889 \ 0.3556 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3.2 \ -0.8 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0 \ 0 \end{pmatrix})</td>
</tr>
<tr>
<td>4</td>
<td>(\begin{pmatrix} 1.6 \ 1.6 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 1.7778 \ 0.7111 \end{pmatrix})</td>
<td>(\begin{pmatrix} 2 \ 1 \end{pmatrix})</td>
<td>(\begin{pmatrix} -0.8 \ 2.2 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0.8 \ 0 \end{pmatrix})</td>
</tr>
<tr>
<td>5</td>
<td>(\begin{pmatrix} 2.4 \ 1.6 \ 0 \end{pmatrix})</td>
<td>(\begin{pmatrix} 1.4222 \ 0.8889 \end{pmatrix})</td>
<td>(\begin{pmatrix} 1 \ 1 \end{pmatrix})</td>
<td>(\begin{pmatrix} 3.2 \ 0.2 \end{pmatrix})</td>
<td>(\begin{pmatrix} 0 \ 0 \end{pmatrix})</td>
</tr>
</tbody>
</table>
Table 4.2: The first five UNA iterations of system (4.3)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$s^{(i)}$</th>
<th>$\hat{x}^{(i)}$</th>
<th>$x^{(i)}$</th>
<th>$r^{(i)}$</th>
<th>$d^{(i)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0)</td>
<td>(0 1/2)</td>
<td>(0 1)</td>
<td>(-1/4 3/4 0)</td>
<td>(1/4 0 0)</td>
</tr>
<tr>
<td>1</td>
<td>(1/4)</td>
<td>(0 3/8)</td>
<td>(0 0)</td>
<td>(3/4 -1/4 0)</td>
<td>(0 1/4 0)</td>
</tr>
<tr>
<td>2</td>
<td>(0)</td>
<td>(0 1/2)</td>
<td>(0 1)</td>
<td>(-1/4 3/4 0)</td>
<td>(1/4 0 0)</td>
</tr>
<tr>
<td>3</td>
<td>(1/4)</td>
<td>(0 3/8)</td>
<td>(0 0)</td>
<td>(3/4 -1/4 0)</td>
<td>(0 1/4 0)</td>
</tr>
<tr>
<td>4</td>
<td>(0)</td>
<td>(0 1/2)</td>
<td>(0 1)</td>
<td>(-1/4 3/4 0)</td>
<td>(1/4 0 0)</td>
</tr>
</tbody>
</table>
Chapter 5
New Approach

LINEAR PROGRAMMING (LP) [42] is a field in mathematical optimization. It involves the optimization of a linear function constrained by a linear equation system. Gupta [8] dismisses the use of LP, on the basis that an optimal solution is not sought and that different techniques have to be used if the system has integer constraints. Still, we would like to have a new discussion on the topic of LP, since we believe that its dismissal is not justified. Therefore, in this section we will look into the simplex method, which is probably the most widely used method in LP, and has been in the focus of research since it was first developed in the 1940s. In the end of this section we will discuss a general method for solving mixed integer problems, based on either iterative or recursive application of the simplex method. [46]

Let us consider the standard form of LP,

\[
\begin{align*}
\text{minimize} \quad & z = (c_1, \ldots, c_n)^T x \\
\text{subject to} \quad & Ax = b \\
& x > 0,
\end{align*}
\]

(5.1)

where \( A \) is an \( m \times n \) matrix with \( m \leq n \). The foundations of LP are based on two important facts [42]. First, if (5.1) has a finite optimal solution, then it is always a corner point (extreme point) of the polyhedron defined by the linear constraints in \( Ax = b \). Second, each corner point corresponds to an invertible \( m \times m \) basis matrix \( B \), which consists of any \( m \) linear independent columns of \( A \). Due to degeneracy [42] two bases \( B_i \) and \( B_j \) may correspond to the same corner point.

Based on these two facts, the simplex method enumerates all bases until an optimal value is found. The worst-case scenario is when the simplex method has to enumerate all possible bases in order to find the optimal solution. Therefore, the theoretical upper bound on the number of iterations is

\[
\binom{n}{m} = \frac{n!}{m!(n-m)!}.
\]

47
Yet, it was not until 1972 that Klee and Minty showed that worst-case problems of arbitrary size exist [42]. Moreover, practical results show that the simplex needs between $m$ and $3m$ iterations to find a solution for problems on standard form.

## 5.1 Simplex vs. UNA

The iteration complexity of the simplex method is more difficult to estimate than the UNA. Not considering periodic refactorization, which is necessary for avoiding accumulation of round-off errors, the operations performed in a simplex iteration are simple matrix and vector calculations, with a total cost of $O(mn)$. This is the same as the cost for the UNA. However, the cost of periodic refactorization is $O(m^3)$ arithmetic operations. Therefore, the cost of a simplex iteration can be as high as $O(m^3 + mn)$ operations.

Even though the simplex method has a much higher iteration cost it is more general in its definition. From the algorithmic description of UNA we recall that it is required that the coefficient matrix $A$ in $Ax \geq b$ has full column rank. This is because the least square solution is calculated using the inverse of $A^T A$, which exists if and only if $A$ has full column rank. If $A$ does not satisfy this criterion a new partition $A'$ of $(A|S)$ must be created, such that $A'$ has full column rank, that is, we must construct a matrix $A'$ of $n$ columns selected from $(A|S)$ such that $A'$ has full column rank.

The procedure of finding an initial coefficient matrix with full column rank is called finding an initial basis. How Gupta et al. handles this complex task is not clear. The two-phase simplex [42], on the other hand, formulates a so-called phase-one problem, the only purpose of which is to find an initial basis that is used in the original (phase-two) problem. In other words, the first phase finds an initial basis (initial solution) by applying the simplex method to a phase-one problem, and the second phase finds the optimal solution by yet another application of the simplex method but now on the original problem. Since we are not interested in optimality it is sufficient to use only the first phase of the two-phase simplex as a constraint solver for a TDG.

Table 5.1 shows the results of when we ran equation system (4.1) and (4.2) through our implementations of UNA and the phase-one simplex method. Note that both systems have an upper bound of $\frac{3!}{2!1!} = 3$ iterations in simplex method.
5.2 Benchmark

Table 5.1: A comparison of the iteration count between UNA and the simplex method.

<table>
<thead>
<tr>
<th>System (4.1)</th>
<th>UNA</th>
<th>Simplex</th>
</tr>
</thead>
<tbody>
<tr>
<td>System (4.2)</td>
<td>1118</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Statistics from solving 18504 random (consistent) equation systems.

<table>
<thead>
<tr>
<th></th>
<th>UNA</th>
<th>Simplex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>188</td>
<td>13.6</td>
</tr>
<tr>
<td>Std. dev.</td>
<td>200</td>
<td>4.44</td>
</tr>
</tbody>
</table>

5.2 Benchmark

A more thorough comparison between the two methods is presented in table 5.2, which shows the average number of iterations for 18504 random consistent equation systems, on the standard form of UNA (3.1), with sizes $m \times n$, where $1 < n < m \leq 20$. The average value of $m$ and $n$ was $m \approx 13.7$ and $n \approx 9.7$. All equation systems had full column rank.

Not only does the UNA show a much higher average iteration count, but it also has an extremely high average standard deviation, which means that the number of iterations for equal-sized matrices vary a lot. However, this is expected since UNA is proved to be unbounded. We see that the simplex method performs more than 13 times better on the average number of iterations, which is approximately equal to $\overline{m}$. The fact that the standard deviation is 45 times for the simplex method shows that the number of iterations per problem size is much more stable compared to UNA.

What must be taken into account here is that the simplex method has a much higher iteration cost, but then again, the conditions of the experiment where overwhelmingly favorable for the UNA, since all equation systems had full column rank. Therefore, it is not justified to compare the actual iteration count from perspectives other than stability and boundedness.

As we mentioned earlier, the simplex method cannot handle integer constraints on the variables. For this we can use a method called branch-and-bound (BB) [43]. If the simplex solution violates an integer constraint, BB cuts off regions of the polyhedron, in order to obtain an integer corner
point. It may sound cumbersome to use two different methods, one for finding real solutions and one for (mixed) integer solutions. But, since BB can use the simplex internally, it can be used to solve problems with or without integer constraints. The latter is merely a special case of the BB.

A drawback with both BB (standard implementation) and UNA occurs when an integer solution is sought and the constraints represent an unbounded polyhedron with no integer solutions. The two algorithms will never terminate with the empty solution. Informal experiments, however, show that a cutting-plane method such as Gomory cuts [43] in combination with branch-and-bound seems to solve this problem.
Chapter 6
Conclusions and Future Work

— Finally, in conclusion, let me say just this.

Peter Sellers

Una is highly dependent on the composition of the input. We have seen that it is very easy to construct systems that make the iteration count increase drastically: system (4.1) requires more than 300 times the number of iterations of the upper bound of the simplex method to be solved. This work shows that there is no upper bound on the number of iterations for una. The practical results presented in chapter 5, which are based on solving a large number of random equation systems, show that the unboundedness is not only of theoretical interest.

What’s more, if the columns of \( A \) are linearly dependent, an initial basis other than \( A \) must be found in order to meet the least square criterion. In other words, \( A^T A \) must be invertible. If it is not, additional iterations for finding a feasible starting matrix must be performed. The simplex method, on the other hand, can be used to find an initial partition. Since we are not looking for an optimal solution, we are done once the initial basis is found.

Furthermore, observation 4.5 shows that under some circumstances una does not find a solution. In contrast, the simplex method together with branch-and-bound definitely finds one.

The conclusion is that a classical optimization algorithm such as the simplex, in combination with branch-and-bound and/or a cutting-plane algorithm are far superior to una.

6.1 Future work

Although the results in this work point out some discouraging features of una, it may still be a suitable algorithm in general. Therefore, more
Chapter 6 Conclusions and Future Work

thorough analysis and experiments can be performed to compare it with other well known linear optimization techniques, e.g. the simplex method. It is of particular interest in evaluating the performance on mixed integer problems.

Another alternative is to see if it is possible to predict when UNA will move into these undesired states, and in those cases use some alternative algorithm. There may also be ways to modify the algorithm in a way that will give better performance.

Furthermore, it is of great interest to remove the severely constraining condition requiring the coefficient matrix to have of full column rank.
Part II

Specification-based Test
Data Generation
Chapter 7

Introduction to Specification-based Test Data Generation

— It is easier to change the specification to fit the program than vice versa.

Alan Perlis

Specifications determine what the program ought to do, therefore using them in software testing is advantageous. In specification-based testing test data is generated from the specification. The program is viewed as a black-box, where focus is on inputs, outputs, and specified behavior, that is, what is implemented. Hence, tests generated from specifications have different objectives than tests generated from programs — specification-based tests allow testing for missing functionality, while program-based tests give us the opportunity of finding out if the program does something not specified [47].

Specification-based test data generation is subject to almost the same architectural model as program-based generation (see figure 7.1). However, specifications do not have control-flow in the same sense as programs, therefore we have omitted the path selector. Additionally, the given specification is used as an oracle for validating the outcome of the generated tests. In this way we have constructed a fully automatic testing tool.

The specification-based generator uses a specification-based criterion to partition the specification into subdomains (paths) that are sampled for test data. Since specifications are often not executable it is not possible for the constraint generator to use actual or symbolic execution, as discussed in section 2.1. Instead, constraints can be generated by symbolic
Figure 7.1: A specification-based testing tool can use the specification as an oracle for expected result.

manipulation of the specification. Many specification language do not have side effects. This simplifies the constraint generation phase greatly.

The constraint solving techniques presented in section 2.2, on the other hand, are directly applicable for specifications.

## 7.1 Specification Languages

Specification languages arose from a need to have unambiguous and concise specifications [48]. A formal notation with well-defined semantics allows for a specification to be dealt with in terms of mathematical reasoning. Therefore, formal specifications are necessary for the automation of software development. Specifically, fully automated testing is only possible when there is a formal specification. Not only can we derive test cases from the specification, but we can also use the specification as an oracle to evaluate test results.

Andrew Harry writes in his book [48] on formal methods that formal methods and specification languages are probably in the stage that programming languages were in the 1960’s. At that time languages were cryptic and had little tool support. This means that the use of the specification languages of today is limited to experts.

A specification must be maintained and updated during all phases in software development, and not only by the experts in analysis, architecture and design. Some of the parties involved in the development process are: customers, analysesists, architects, designers, programmers, testers, and maintainers. It is important that the specification is understood by
There are numerous specification languages. Among the most well known are VDM-SL, Z, B and Estelle. Various languages differ in their style of specification and their suitability for special kinds of systems such as real time systems or finite automata. Many of these languages are very complex and perhaps not appropriate in all phases of the development process. Simpler specification languages have been developed, which more closely match actual implementation languages [49,50].

### 7.1.1 Styles of Specification Languages

Specification languages can be roughly divided into the styles of model-based and property-based [48]. A common property of both styles is that languages are classified as implicit, explicit, or somewhere in between. Being implicit means that operations are expressed in terms of *what* they do, while being explicit means expressing *how* they do it. Explicit specifications are closer to an actual implementation, and thus have a higher degree of executability.

Any language can be used as a specification language. If a traditional imperative programming language such as Ada, Pascal, or C is used, the specification will be truly explicit, since all operations are executable. There has been debate about whether specifications should be executable or not [51,52]. An explicit specification may constrain an implementation to a particular design due to its algorithmic style. Conversely, an implicit specification will leave the algorithmic solution open. As an example of an unexecutable (implicit) specification, consider a function returning the integer square root of its argument, where \( x, y \in \mathbb{N} \).

\[
\sqrt{x} = y : y^2 \leq x < (y+1)^2
\]

Here there is no algorithmic restriction on how to implement \( \sqrt{x} \). Instead, a relationship between input and output is specified.

**Model-based**

Model-based languages consider programs as abstract machines with internal states. Operations on these machines are used to model transitions between states. The square root function above is a typical example of a model-based specification.

Compared to property-based languages, model-based specifications are closer to an actual implementation. Examples of model-based languages are VDM-SL, Z, B (AML), JML and iContract [48–50,53].
Chapter 7 Introduction to Specification-based TDG

There is a great variation in syntax among these languages. The z language, for instance, uses an advanced mathematical notation, which for the novice is difficult to understand. On the other end of the scale we have languages such as iContract, which is a specification language based on Java. Below we see an excerpt from an iContract model of a stack.

```java
public class Stack {
    private Object[] elements;
    private int count;

    /**
     * @ensure count == count@pre+1 && |
     *      (forall int i in 0 .. size()-2 |
     *       elements[i] == elements[i]@pre) && |
     *      elements[count-1] == e
     */
    public void push(Object e) ...

    /**
     * @require !isEmpty()
     * @ensure count == count@pre-1 && |
     *      (forall int i in 0 .. size() |
     *       elements[i] == elements[i]@pre) && |
     *      return == element[count]@pre
     */
    public Object pop() ...

    /**
     * @ensure return == (count == 0)
     */
    public boolean isEmpty() ...
}
```

The keyword @ensure is used to model a method’s postcondition, a condition that is guaranteed to hold after execution of the method. For instance, the method isEmpty() returns true if and only if the number of elements count is 0. The keyword @pre is used to refer to the context before execution of the method.

A method’s precondition, modeled by @require, is used to indicate under what circumstances the postcondition holds. For example, pop()
states that only when the stack is not empty is the postcondition guaranteed to hold. The behavior of pop() when the stack is empty is unspecified. A missing @require such in the case of isEmpty() is equivalent to @require true, that is, the postcondition is always applicable.

**Property-based**

This specification style hosts languages that are axiomatic and algebraic. As described by Hayes et al. [54] a property-based specification language would model a stack with axioms such as

\[
\text{pop(push(x, s))} = s
\]

rather than specifying what will happen with stack’s internal state upon a pop- or push-operation. OBJ is an example of an algebraic language.

**Concurrency-based**

A concurrency-based style is not a unique style per se, but can be applied to both model-based and property-based styles. Languages in this style specialize in modeling concurrent programs and processes.

In this work we focus on model-based languages since these are better suited for modeling programs at the unit level. Being closer to actual implementation, it is realistic to assume that programmers implementing the unit better understand this type of specification.

### 7.2 Adequacy Criteria for Specifications

The set of all possible inputs for a program is called the input domain. Exhaustive testing exercises the whole input domain. Practically, however, this is not feasible since the input space is often extremely large, thus making better adequacy criteria important as this will reduce the number of tests.

According to one of the most fundamental principle of testing, called partition testing, an adequacy criterion divides the input space of the subject, e.g. a program or a specification, into subdomains, among which one or more representatives are selected as inputs. The rationale behind this is that each representative tests its subdomain equally well. The strength of the generated input set, that is, its effectiveness of detecting faults, depends on how well the criterion partitions the input domain.
Chapter 7 Introduction to Specification-based TDG

if \((a > 0 \lor b < 5)\) then
\[ r = a + b \]
else
\[ r = 2a \]

\[ \begin{cases} \quad (a > 0 \lor b < 5) \land r = a + b \\ \quad (a \leq 0 \lor b \geq 5) \land r = 2a \end{cases} \]

(a)

(b)

\[ \begin{cases} a > 0 \land b < 5 \land r = a + b \\ a \leq 0 \land b < 5 \land r = a + b \\ a > 0 \land b \geq 5 \land r = a + b \\ a \leq 0 \lor b \geq 5 \land r = 2a \end{cases} \]

(c)

Figure 7.2: The granularity of the adequacy criteria decides how the specification is interpreted.

The stronger a criterion is, the more details of the subject is taken into account. In general this means that more inputs are needed to meet the criteria, which implies more work. In this way adequacy criteria play a central role in input generation, since it ultimately decides what is to be tested and how thorough it will be tested.

Adequacy criterion are sometimes called stopping criterion or coverage criterion in the literature. The latter name actually refers to a subset of adequacy criteria that are expressed as covering, for instance, the subdomains of partition or the nodes of a flow graph. In this work all of these names are used more or less synonymously.

The interpretation of a specification is directly coupled to the adequacy criterion used. Consider the hypothetical specification in figure 7.2(a), for which two different interpretations of the same specification are presented. The criterion used in figure 7.2(b) generates a partition of two subdomains, whereas the criterion in figure 7.2(c) is more precise and generates four. Viewing the specification as a graph, an adequacy criterion states what nodes and edges are considered vital for the graph. The granularity of the adequacy criterion affects how the specification is interpreted.

Most of the current specification-based testing techniques use manual methods that cannot be easily automated [55]. However, Dick and Faivre [56] describe a technique for automatic partitioning of VDM-SL
7.2 Adequacy Criteria for Specifications

specifications. The specification is transformed into disjunctive normal form (DNF), where each conjunctive clause corresponds to a subdomain. For example, the expression $a \vee b$ is transformed into three disjoint cases $\neg a \land b$, $a \land b$, and $a \land \neg b$.

Model-based specifications can be quite flat, since powerful operators can be used to model behavior. Therefore, Stocks and Carrington [57] propose a partitioning technique called domain propagation. They mean that if the powerful operators are not accounted for the resulting partition may be too coarse.

For example, using only DNF transformation, an expression $x \geq 0$ which is already on DNF, will not be further partitioned. Domain propagation, however, recognizes the fact that $x \geq 0$ can be partitioned into $x = 0$ and $x \geq 1$, if $x$ is integer.

Weyuker et al. [58] use mutation analysis, which is easily automated, to assess adequacy of a set of test cases derived from boolean specifications. Various strategies used in test generation are examined in terms of the capability of detecting certain faults, such as negation of variable etc.

Meudec [9] presents an automatic technique, based on the work of by Dick and Faivre, and Stock and Carrington, for generating tests from specification. Here, the specification is partitioned into subdomains by symbolic rewrites of the specification according to a set of rules. For instance, assume a specification stating $(a \vee b) \land c$, where the variables are not necessarily are boolean, but arbitrary predicates. According to Meudec’s rules, which are covered in detail in chapter 8, the expression is partitioned into the set of subdomains shown in 7.3.

Meudec’s motivation for deriving this particular set of subdomains is based on simple reasoning about how to interpret the operators $\land$ and $\lor$. A conjunctive expression such as $a \land b$ can only give rise to one equivalence class (subdomain), namely the one where $a$ is true and $b$ is true.

Similarly, an expression $a \lor b$ can only be instantiated to true in three ways. Therefore it is partitioned into the following set of subdomains.

\[
\begin{align*}
\{ a \land b \} \\
\{ \neg a \land b \} \\
\{ a \land \neg b \}
\end{align*}
\]

Meudec’s partitioning technique (with modification) has been implemented by Atterer in a program called vdmpart [59], which partitions specifications written in VDM-SL. Atterer did not consider partition minimization but focused on reducing the size of the so-called complete partition.
Figure 7.3: Partitioning \((a \lor b) \land c\) according to Meudec’s rules gives a set of three subdomains. A truth table corresponding to the subdomains gives a possible input variable assignment.

Offutt et al. [55] define a similar criterion to Meudec’s, called full predicate coverage. It is defined as: for each predicate \(P\) on each transition (operation), the set of tests \(T\) must include tests that cause each clause \(c\) in \(P\) to result in a pair of outcomes where the value of \(P\) is directly correlated with the value of \(c\).

For example, given the same expression \((a \lor b) \land c\) their criterion creates six subdomains. The difference between Meudec’s criterion and full predicate coverage is that the latter seems to consider the domain of false behavior as well.

For executable specifications or specifications modeled as state machines it is possible to use program-based criteria for specifications. Some of the most common program-based criteria include control-flow coverage such as statement coverage, branch coverage and path coverage, and various types of data-flow coverage.

However, since programs do not necessarily reflect the control-flow nor the data-flow of a corresponding specification, program-based criteria may not be an adequate choice. The next part of the thesis explores whether branch coverage can be used as criteria for specifications written in Java.

More information on both specification-based and program-based coverage is presented in a survey by Zhu et al [60].

The simplicity of deriving a partition using Meudec’s technique makes it very interesting. However, for more complex specifications Meudec’s partitioning rules produce an exponential number of subdomains, which in turn limits its usefulness. Therefore, Meudec developed an algorithm that removes redundant subdomains, producing a so-called minimal partition. However, this work shows that his algorithm is incorrect. This is the subject in the rest of part II, where we analyze Meudec’s work and
7.2 Adequacy Criteria for Specifications

\[
\begin{array}{cccc}
(a \lor b) \land c & \equiv & r \\
T & F & T & T \quad \leftarrow \text{case 3} \\
F & F & F & F \\
F & T & T & T \quad \leftarrow \text{case 2} \\
F & F & F & F \\
T & T & T & T \quad \leftarrow \text{case 1} \\
T & T & F & F \\
\end{array}
\]

Figure 7.4: Full predicate coverage of \((a \lor b) \land c\) gives a truth table contain not only Meudec’s cases but also three additional specifying the false behavior.

contribute with two new methods for finding minimal partitions.

Both methods require that the partition to minimize is dependence-free. Therefore, an algorithm for removing dependence in partitions is also developed. The technique is based on so-called partial expansion.

The rest of part II is organized as follows. In chapter 8 Meudec’s automatic partitioning technique is described. Here, basic concepts regarding automatic partitioning and partition composition is covered. In order to get a more consistent and expressive notation, some of the material introduced is new and some is not entirely consistent with that of Meudec’s.

In chapter 9 previous work by Meudec in redundancy analysis is explored. The chapter ends with an analysis of Meudec’s proposed algorithm for finding a minimal partition proving it to be incorrect.

In chapter 10 an algorithm to remove partition dependence through partial expansion is developed. In chapter 11 a new minimal partition method, referred to as the Set-cover method, is constructed. The method is based on integer programming. The algorithm uses dependence-free partitions to improve performance through column generation.

In chapter 12 an alternative approach for finding a minimal partition is used. This method, called the Path-cover method, models the problem as a network problem.

In chapter 13 an improvement to the partial expansion algorithm presented in 10 is made. This results in more compact graphs, which are better suited for the Path-cover method.

Chapter 14 discusses alternatives when even minimal partitions are too large. Finally, in chapter 15 conclusions and future work are presented.
Chapter 8
Partition Testing

— I wonder sometimes if manufacturers of foolproof items keep a fool or two on their payroll to test things.

Alan Coren

Meudec presents in his dissertation [9] an approach that automatically generates tests from a VDM-SL specification using an automatic partitioning strategy. His strategy can be summarized into the following phases.

1. Systematic partitioning. Output of this phase is a set of subdomains.

2. Redundancy analysis. Subdomains considered redundant are removed. Ultimately a minimal set of subdomains is found, a so-called minimal partition.

3. Sampling. The subdomains are sampled for test data.

Several practical difficulties are encountered when implementing this approach. One of those problems is how to implement the solver needed for consistency checking and sampling. Another, to which this work is devoted, is the redundancy analysis.

This chapter builds up the theory required to understand Meudec’s work. It explores how specification expressions are transformed into partitions using Meudec’s rule-based partitioning function. Typically, a VDM-SL specification is composed of different constructs: if expressions, logical expressions, arithmetic relations, function calls etc. However, as far as redundancy analysis is concerned, it is not required that all details regarding the partitioning rules are covered.
Chapter 8  Partition Testing

In fact, as will be clear in successive chapters, a partition can be seen as a directed acyclic graph arranged in levels, where each node corresponds to a logic clause (see figure 8.1) such as $a + b \leq c$ or $f(a, b) > 0$. A path through the graph corresponds to a path constraint, that is, a conjunction of the clauses of the path.

Thus, the methods developed in this work are applicable not only to Meudec’s partition method, but to any partition method that can be described in a partition graph.

There are two reasons for presenting details of Meudec’s method for generating partitions, and not heading directly for the notion of partition graphs. First, it is needed to understand the workings of his minimal partition method. Second, it is needed to understand what partition graphs look like.

Therefore, without loss of generality, only a subset of the rules is presented, assuming a specification language consisting of if-expressions, logical expressions ($\land, \lor, \forall, \exists$), arithmetic expressions, relational expressions (e.g. $a + b < c$ or $a = b$), and variables (integer, real and boolean).

In this chapter Meudec’s work is revised and extended in order to get a clearer nomenclature and a better foundation for subsequent sections. Therefore this section is also of great importance for readers familiar with Meudec’s work.

8.1 Systematic Partitioning

Meudec defines a function $P(expr) = \{s_1, \ldots, s_n\}$ that partitions a given expression into a set of subdomains (also called compound domain). The partition function is defined as set of rules describing how partitions are generated from various VDM-SL expressions. Below is a list of the most
8.1 Systematic Partitioning

basic rules.

\[
\begin{align*}
P(A) &= \{\langle A \rangle \} \quad \text{when } A \text{ is boolean variable} \quad (8.1) \\
P(\neg A) &= \{\langle \neg A \rangle \} \quad \text{when } A \text{ is a boolean variable} \quad (8.2) \\
P(\neg A) &= P(\text{dev}(\neg A)) \quad \text{when } A \text{ is not a boolean variable} \quad (8.3) \\
P(A \land B) &= P(A) \times P(B) \quad (8.4) \\
P(A \lor B) &= \begin{cases} P(A) \times P(\neg B) \\
P(A) \times P(B) \\
P(\neg A) \times P(B) \end{cases} \quad (8.5)
\end{align*}
\]

The function \text{dev} develops its given argument one level. For example, 
\text{dev}(\neg(A \lor B)) = \neg A \land \neg B. The operator \times is called the full-combination operator and is formally defined in definition 8.6. For now, however, it is sufficient to think of it as similar to ordinary multiplication. For example, assume that \(a, b, c, \) and \(d\) are boolean variables.

\[
\begin{cases}
\langle a \rangle \\
\langle b \rangle
\end{cases} \times \begin{cases}
\langle c \rangle \\
\langle d \rangle
\end{cases} = \begin{cases}
\langle a \rangle \times \langle c \rangle \\
\langle a \rangle \times \langle d \rangle \\
\langle b \rangle \times \langle c \rangle \\
\langle b \rangle \times \langle d \rangle
\end{cases} = \begin{cases}
\langle a \land c \rangle \\
\langle a \land d \rangle \\
\langle b \land c \rangle \\
\langle b \land d \rangle
\end{cases}
\]

**Definition 8.1.** An atomic domain \(\langle p_1 \land \ldots \land p_n \rangle\) is a conjunction of predicates defining the extent of the domain.

When the full-combination operator is applied on two atomic domains they are merged by a logic conjunction to form a new atomic domain. The process of applying the \times operator is called expansion. Particularly, if the operator is applied on two atomic domains it is called merging. For example, when partitioning \(a \land b\) one obtains:

\[
P(a \land b) = P(a) \times P(b) = \{\langle a \rangle \} \times \{\langle b \rangle \} = \{\langle a \rangle \times \langle b \rangle \} = \{\langle a \land b \rangle \}.
\]

**Definition 8.2.** A domain on the form \(S = \{s_1, \ldots, s_n\}\), where \(s_i\) are subdomains is called compound domain. If the compound domain only consists of a single atomic subdomain, that is, \(\{\langle p \rangle \}\) it is often simplified to \(\langle p \rangle\).
Chapter 8 Partition Testing

Consider the partitioning of the following boolean expression \( a \land \neg(b \land c) \).

\[
P(a \land \neg(b \land c)) = P(a) \times P(\neg(b \land c))
\]
\[
= \langle a \rangle \times P(\neg(b \land c))
\]
\[
= \langle a \rangle \times P(\neg b \lor \neg c)
\]
\[
= \langle a \rangle \times \left\{ \begin{array}{l}
P(\neg b) \times P(c) \\
P(\neg b) \times P(\neg c) \\
P(b) \times P(\neg c)
\end{array} \right\}
\]
\[
= \langle a \rangle \times \left\{ \begin{array}{l}
\langle \neg b \rangle \times \langle c \rangle \\
\langle \neg b \rangle \times \langle \neg c \rangle \\
\langle b \rangle \times \langle \neg c \rangle
\end{array} \right\}
\]

Pause the process for a moment. The above expression is referred to an *unexpanded* partition, that is, the partition function has completed, but there has not yet been an application of the full-combination operator. Normally in Meudec’s partitioning strategy, all full-combination operators are expanded as the partitioning proceeds. Later in this work, it will be clear that it is not only of interest to view the partition in fully expanded form, but also in unexpanded (or partially expanded) form. Continuing in with the expansion process one obtains:

\[
\langle a \rangle \times \left\{ \begin{array}{l}
\langle \neg b \rangle \times \langle c \rangle \\
\langle \neg b \rangle \times \langle \neg c \rangle \\
\langle b \rangle \times \langle \neg c \rangle
\end{array} \right\} = \langle a \rangle \times \left\{ \begin{array}{l}
\langle \neg b \land c \rangle \\
\langle \neg b \land \neg c \rangle \\
\langle b \land \neg c \rangle
\end{array} \right\}
\]

Finally, a fully expanded partition consisting of three atomic subdomains is obtained. As a comparison, the partially expanded partition obtained just before the last step, consists of four atomic subdomains.

\[
\langle a \rangle \times \left\{ \begin{array}{l}
\langle \neg b \land c \rangle^2 \\
\langle \neg b \land \neg c \rangle^3 \\
\langle b \land \neg c \rangle^4
\end{array} \right\}
\]

Now that we are more acquainted with how the rule-based partitioning
process works, we present the remaining rules that are used in this work.

\[
P(if \ B \ then \ E_1 \ else \ E_2) = \begin{cases} 
P(B) \times P(E_1) \\
P(\neg B) \times P(E_2) \end{cases} \quad (8.6)
\]

\[
P(x = y) = \begin{cases} 
\langle x = y \rangle 
\end{cases} \quad (8.7)
\]

\[
P(x \neq y) = \begin{cases} 
\langle x = y + \delta \rangle \\
\langle x > y + \delta \rangle \\
\langle x = y - \delta \rangle \\
\langle x < y - \delta \rangle 
\end{cases} \quad (8.8)
\]

\[
P(x \geq y) = \begin{cases} 
\langle x = y \rangle \\
\langle x > y \rangle 
\end{cases} \quad (8.9)
\]

\[
P(x \leq y) = \begin{cases} 
\langle x = y \rangle \\
\langle x < y \rangle 
\end{cases} \quad (8.10)
\]

\[
P(x > y) = \begin{cases} 
\langle x = y + \delta \rangle \\
\langle x > y + \delta \rangle 
\end{cases} \quad (8.11)
\]

\[
P(x < y) = \begin{cases} 
\langle x = y - \delta \rangle \\
\langle x < y - \delta \rangle 
\end{cases} \quad (8.12)
\]

Here \(\delta\) means the smallest smallest increment of the type of \(x\). For example, when \(x\) is integer \(\delta = 1\).

Furthermore, consider how to partition a universally or existentially quantified expression, for example, for all \(x \in D\) such that constraint \(C\) holds. These types of expression are partitioned as follows. Let \(P(C) = \{\langle c_1 \rangle, \ldots, \langle c_n \rangle\}\) and let \(T_i = \{\langle \exists x \in D : c_i \rangle, \langle \forall x \in D : \neg c_i \rangle\}\).

\[
P(\forall x \in D : C) = \begin{cases} 
\langle D = \emptyset \rangle \\
\langle \forall x \in D : C \rangle \times T_1 \times \cdots \times T_n 
\end{cases} \quad (8.13)
\]

\[
P(\exists x \in D : C) = \begin{cases} 
P(D \neq \emptyset) \times P(\forall x \in D : C) \\
\langle \exists x \in D : C \rangle \times T_1 \times \cdots \times T_n 
\end{cases} \quad (8.14)
\]

For example, derive the partition of \(x > 0 \lor y < 5\), where \(x\) and \(y\) are
Chapter 8 Partition Testing

integers. We have $P(x > 0 \lor y < 5)$

$$
P(x > 0) \times P(y \geq 5) \quad P(x > 0) \times P(y < 5) \quad P(x \leq 0) \times P(y < 5) = \left\{ \begin{array}{c}
\{x = 1\} \times \{y = 5\} \\
\{x > 1\} \times \{y > 5\} \\
\{x = 1\} \times \{y = 4\} \\
\{x > 1\} \times \{y < 4\} \\
\{x = 0\} \times \{y = 4\} \\
\{x < 0\} \times \{y < 4\}
\end{array} \right\} \quad \text{(8.15)}$$

The resulting partition consists of 12 atomic subdomains. Selecting one candidate from each subdomain gives 12 test cases.

In the next example $\forall x \in D : x \geq 0$ is partitioned, where $D$ is set of integers. The formula states that all elements of $D$ are greater than or equal to zero. Given that $P(x \geq 0) = \{x = 0, x > 0\}$ one can derive a partition for the complete expression $P(\forall x \in D : x \geq 0) =$

$$
\left\{ \begin{array}{c}
D = 0 \\
\forall x \in D : x \geq 0 \\
\forall x \in D : x \geq 0 \\
D = 0
\end{array} \right\} \times \left\{ \begin{array}{c}
\exists x \in D : x = 0 \\
\forall x \in D : x \neq 0
\end{array} \right\} \times \left\{ \begin{array}{c}
\exists x \in D : x > 0 \\
\forall x \in D : x \leq 0
\end{array} \right\}
$$

$$
= \left\{ \begin{array}{c}
D = 0 \\
\forall x \in D : x \geq 0 \\
\forall x \in D : x \geq 0 \\
D = 0
\end{array} \right\} \times \left\{ \begin{array}{c}
\exists x \in D : x = 0 \land \exists x \in D : x > 0 \\
\exists x \in D : x = 0 \land \forall x \in D : x \leq 0 \\
\forall x \in D : x \neq 0 \land \exists x \in D : x > 0 \\
\forall x \in D : x \neq 0 \land \forall x \in D : x \leq 0
\end{array} \right\}
$$

$$
= \left\{ \begin{array}{c}
D = 0 \\
\forall x \in D : x \geq 0 \land \exists x \in D : x = 0 \land \exists x \in D : x > 0 \\
\forall x \in D : x \geq 0 \land \exists x \in D : x = 0 \land \forall x \in D : x \leq 0 \\
\forall x \in D : x \geq 0 \land \forall x \in D : x \neq 0 \land \exists x \in D : x > 0 \\
\forall x \in D : x \geq 0 \land \forall x \in D : x \neq 0 \land \forall x \in D : x \leq 0
\end{array} \right\}
$$

70
Definition 8.3. A terminal domain is an atomic domain that has not been obtained by an application of the full-combination operator.

Intuitively, terminal domains are created as a consequence of a termination of the recursion of the partition function. For example, consider the rule for \( P(x \geq y) = \{\langle x = y \rangle, \langle x > y \rangle \} \). It creates two terminal domains. The rule for universally quantified expressions, on the other hand, produces \( 2^n + 2 \) terminal domains, since there are \( n \) expressions of the form of \( T_i \) containing two terminal domains each, in addition to \( \langle D = \emptyset \rangle \) and \( \langle \forall x \in e : expr \rangle \).

8.2 Minimal Partitions

Because of the exponential nature of the partitioning rules one can, in general, expect that the number of generated subdomains is large. In turn, this may have an impact on the usefulness of Meudec’s method. The number of subdomains can be too large to be considered feasible to process. Remember that each subdomain is to be sampled (solved) by the constraint solver in order to find a test data candidate. Typically, constraint solving problems are very difficult.

Meudec points out that according to the principles of domain testing [4] several of the generated subdomains are in fact redundant and can be removed. For example, consider the partition of the expression \( a \lor b \), where \( a \) and \( b \) are independent boolean variables.

\[
P(a \lor b) = \begin{cases} P(a) \times P(\neg b) \\ P(a) \times P(b) \\ P(\neg a) \times P(b) \end{cases} = \begin{cases} \langle a \rangle \times \langle \neg b \rangle \\ \langle a \rangle \times \langle b \rangle \\ \langle \neg a \rangle \times \langle b \rangle \end{cases} = \begin{cases} \langle a \land \neg b \rangle \\ \langle a \land b \rangle \\ \langle \neg a \land b \rangle \end{cases}
\]

Since \( a \) and \( b \) are independent (orthogonal) the subdomain \( \langle a \land b \rangle \) can be considered redundant and be removed. That is, it is not justified from a testing point of view to test all combinations of expressions that are independent.

The notion of redundancy implies that there is a partition \( M \subseteq P \) such that \( M \) contains or covers all important subdomains of \( P \) and for any other covering partition \( M' \subseteq P \) then \( M \) is the smallest, that is, \( |M| \leq |M'| \).

Simply put, what Meudec means by saying a subset \( M' \) covers all important subdomains of \( P \) is that all unique terminals in \( P \) are also in \( M' \) and all combinations of dependent terminals in \( P \) are also in \( M' \). \( M \) is a minimal subset covering \( P \).
Chapter 8 Partition Testing

Meudec replaces each terminal subdomain by a unique label, such that for two terminals \( t_i, t_j \in P \) they have the same label if and only if \( t_i = t_j \).

Exactly how equality of two terminals is decided is not of importance for the rest of this work. Meudec uses a fairly simple definition of equality that is rather complicated to explain (see [13]). A much simpler, and in fact better, choice is to use syntactic equality. Of course, semantic equality would be the best, but that is not possible in the general case.

Consider partition \( P \) given below (\( \land \) has been replaced by comma). Here, the actual predicates have been replaced by labels. A label consists of a letter indicating the dependence class of the predicate and an index. Labels with the same dependence class represent expressions that are mutually dependent.

\[
P = \left\{ \langle a_1 \rangle, \langle a_2 \rangle \right\} \times \left\{ \langle b_1 \rangle, \langle b_2 \rangle \right\} \times \left\{ \langle c_1 \rangle, \langle c_2 \rangle, \langle a_3 \rangle \right\} = \left\{ \langle a_1, b_1, c_1 \rangle, \langle a_1, b_1, c_2 \rangle, \langle a_1, b_1, a_3 \rangle, \langle a_1, b_2, c_1 \rangle, \langle a_1, b_2, c_2 \rangle, \langle a_1, b_2, a_3 \rangle, \langle a_2, b_1, c_1 \rangle, \langle a_2, b_1, c_2 \rangle, \langle a_2, b_1, a_3 \rangle, \langle a_2, b_2, c_1 \rangle, \langle a_2, b_2, c_2 \rangle, \langle a_2, b_2, a_3 \rangle \right\}
\]

For example, \( a_1 \equiv x > 0 \), \( a_2 \equiv y < 5 \), and \( a_3 \equiv x + y = 2 \). Since both \( x \) and \( y \) are used in the same formula \((a_3)\) they are considered dependent. Therefore all three predicates belong to the same dependence class called \( a \). Again, exactly how dependence is established is not of interest. What is of interest is that there is something called dependence and in what way this is handled.

A partition \( M \) that covers the important subdomains of \( P \) must first of all contain all terminals of \( P \), that is \( a_1, a_2, a_3, b_1, b_2, c_1, \) and \( c_2 \) must be represented in \( M \) at least once.

Second, all combinations of dependent terminals in \( P \) must also be in \( M \), that is, the combinations \( a_1 \times a_3 \), and \( a_2 \times a_3 \) must be represented in \( M \) at least once, because these are the only dependent terminals that are combined.

The set of subdomains below fulfills these requirements. It is also the smallest such set and therefore it is a minimal partition (or minimal set of subdomains).
Finding a covering partition from a fully expanded partition is quite simple. Sort the labels of the atomic subdomains in order of dependence class and index. Then replace all combinations of dependent labels with a new label. For example, below the label \( a_{23} \) is a substitute for \( a_2 \land a_3 \).

\[
M = \begin{cases} 
  \langle a_1, b_1, a_3 \rangle \\
  \langle a_1, b_2, c_1 \rangle \\
  \langle a_2, b_1, a_3 \rangle \\
  \langle a_2, b_2, c_2 \rangle \\
\end{cases}
\]

Now, a covering set of subdomains only needs to contain all labels, since there are no longer any dependent terminal that are combined, \( M = \{ \langle a_{13}, b_1 \rangle, \langle a_1, b_2, c_1 \rangle, \langle a_{23}, b_1 \rangle, \langle a_2, b_2, c_2 \rangle \} \).

Shortly, a formal definition of a minimal partition will be given, but before that some additional concepts must be defined. Specifically, the \( \times \)-operator is to be redefined such that whenever dependent labels are combined they automatically form a new label.

**Definition 8.4.** A label is an abstraction of a predicate. Any naming scheme for labels can be used as long as two important concepts are defined. The first is the predicate \( \text{dep}(l_1, l_2) \) that determines whether the two labels are dependent.

The second, is the function \( \text{getLabel}(l_1, l_2) \) that returns a label \( l \) formed from the two labels \( l_1 \) and \( l_2 \) such that \( (l) = \langle l_1, l_2 \rangle \).

The naming scheme for labels used in the examples is restricted to 10 terminal domains per dependence class. However, as just mentioned it can easily be extended.
Definition 8.5. An atomic domain is a set of mutually independent labels, denoted \( \langle l_1, \ldots, l_n \rangle \), where each label is associated with a predicate expression.

Definition 8.6. The full combination operator \( \times \) is then defined as:

1. \( A \times B = \text{merge}(A, B) \), where \( A \) and \( B \) are atomic domains.

   \[
   \text{function merge}(A, B : \text{atomic domain}) : \text{atomic domain} \\
   \begin{align*}
   R &:= B \\
   \text{foreach } a \in A \text{ do} \\
   \quad \text{if } (\exists r \in R : \text{dep}(a, r)) \text{ then} \\
   \quad \quad R &:= (R - r) \cup \{ \text{getLabel}(a, r) \} \\
   \quad \text{else} \\
   \quad \quad R &:= R \cup \{ a \} \\
   \end{align*}
   \]

   \[\text{return } R\]

2. \( A \times B = \{ A \times b \mid b \in B \} \), where \( A \) is an atomic domain and \( B \) is a compound domain.

3. \( A \times B = \{ a \times B \mid a \in A \} \), where \( A \) and \( B \) are compound domains.

   When two dependent, atomic domains are combined the result is a new atomic domain formed by merging the labels of the two domains. Merging is similar to set union, however, two dependent labels are replaced by a new label representing the two. For example,

   \[
   \langle a_5 \rangle \times \langle a_1 \rangle \times \langle a_4 \rangle \times \langle c_1 \rangle \times \langle b_1 \rangle \times \langle a_1 \rangle \times \langle a_0 \rangle
   \]

   \[
   = \langle a_{15} \rangle \times \langle a_4 \rangle \times \langle c_1 \rangle \times \langle b_1 \rangle \times \langle a_1 \rangle \times \langle a_0 \rangle
   \]

   \[
   = \langle a_{145} \rangle \times \langle c_1 \rangle \times \langle b_1 \rangle \times \langle a_0 \rangle
   \]

   \[
   = \langle a_{145}, c_1 \rangle \times \langle b_1 \rangle \times \langle a_0 \rangle
   \]

   \[
   = \langle a_{0145}, b_1, c_1 \rangle
   \]

   As can be seen, the predicate-form and label-form of an atomic domain are isomorphic. The functions \( \text{pred}(a) \) and \( \text{labels}(a) \) returns the predicate and the set of labels for an atomic domain \( a \) respectively.

Definition 8.7. An unexpanded domain composed of either compound domains or terminal domains, that is, all inner-most subdomains are terminal. It is also the result of the partition function.
8.2 Minimal Partitions

A domain of the form \( \{a_1, a_2, \ldots, a_n\} \), where all \( a_i \)'s are atomic domains is called a **fully expanded domain**. A domain that is not unexpanded nor fully expanded is called **partially expanded** domain.

**Definition 8.8.** Let \( S \) be a domain and let \( \text{expand}_F(S) \) return the fully expanded domain of \( S \), then \( \text{labels}(S) \) returns the set of all labels in all subdomains of \( S \).

\[
\text{labels}(S) = \bigcup \{ l \mid s \in \text{expand}_F(S) \land l \in s \}
\]

**Definition 8.9.** A domain \( S = \{s_1, \ldots, s_n\} \), where the \( s_i \)'s are atomic is called **consistent** if and only if none of the subdomains \( s_i \) has the empty solution.

**Definition 8.10.** Let \( P \) be a fully-expanded consistent domain. The positive closure of \( P \) is defined as the set of all sets of \( P \) that cover the labels of \( P \). That is,

\[
P^+ = \{ R \mid R \subseteq P \land \text{labels}(R) = \text{labels}(P) \}.
\]

Similarly, the negative closure \( P^- \) is defined as

\[
P^- = \{ R \mid R \subseteq P \land \text{labels}(R) \subset \text{labels}(P) \}.
\]

**Definition 8.11 (Minimal Partition).** Let \( P \) be a fully-expanded consistent domain then a minimal partition (domain) \( M \) of \( P \) is such that

\[
M \in P^+ \land \forall M' \in P^+: |M'| \geq |M|
\]

where \(|A|\) returns the number of elements in set \( A \).

Don’t worry if the formal definition of minimal partitions is difficult to grasp at first. It took me several months before I fully understood this part of Meudec’s work. Add to this, what I present to you is a much cleaner and more easily understood definition than the one in Meudec’s thesis. In successive chapters, when converting to a partition graph representation things will probably become more clear.

It is normal that some of the subdomains domains of a fully-expanded partition \( P \) are inconsistent. This is similar to the problem in path testing: some of the paths (of all paths) of a program may not be reachable.

It is important to note that if \( P \) contains inconsistent subdomains these must first be removed, otherwise the minimal partition may contain some of those subdomains, which in turn may result in some label not being
Chapter 8 Partition Testing

covered and thus the resulting partition would not be minimal. A method for removing inconsistency from the partition is an indirect result of the partial expansion algorithm that is developed in successive chapters. How inconsistency can be removed is described on page 87.

A final example of a minimal partition concludes this chapter. Consider the partition given below. In fact, it is the labeled version of partition (8.15). A partition where only independent labels are combined cannot contain inconsistent subdomains, unless the terminal subdomains themselves are inconsistent, which it is assumed they are not.

\[
P(x > 0 \lor y < 5) = \left\{ \begin{array}{c}
\{x_1\} \times \{y_1\} \\
\{x_2\} \times \{y_2\} \\
\{x_1\} \times \{y_3\} \\
\{x_2\} \times \{y_4\} \\
\{x_3\} \times \{y_3\} \\
\{x_4\} \times \{y_4\}
\end{array} \right\} = \left\{ \begin{array}{c}
\{x_1, y_1\} \\
\{x_1, y_2\} \\
\{x_2, y_1\} \\
\{x_2, y_2\} \\
\{x_1, y_3\} \\
\{x_1, y_4\} \\
\{x_2, y_3\} \\
\{x_2, y_4\} \\
\{x_3, y_3\} \\
\{x_3, y_4\} \\
\{x_4, y_3\} \\
\{x_4, y_4\}
\end{array} \right\} \quad (8.16)
\]

Then one minimal partition is \( M = \{s_1, s_4, s_9, s_{12}\} \). It is simple to argue that it is minimal. Firstly, the set of all labels for the minimal partition is

\[
\text{labels}(P(e)) = \{x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4\} = \text{labels}(M)
\]

which means that all 8 labels are represented. Secondly, since each atomic subdomain in the complete partition represents exactly one \( x_i \) and one \( y_i \) the smallest partition must contain at least 4 subdomains.
Chapter 9

Meudec’s Minimal Partition Method

— No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

Albert Einstein

Redundancy analysis is the process of reducing the number of subdomains in the complete partition. Naturally, a reduced partition $M'$ must satisfy $M' \subseteq P$. Optimally, we would like to find a minimal partition $M$ such that there is no other partition $M' \subseteq P$ such that $M' \subset M$, where $M'$ covers the set of all labels. In this chapter Meudec’s method for finding a minimal set of subdomains is described. Before doing so, we will first learn about some of the difficulties of the problem by studying approaches, considered by Meudec, that do not work. We will learn that finding a minimal partition is more difficult than Meudec initially anticipated.

The concept of redundant subdomains is defined with respect to a complete partition $P$ and a minimal minimal partition $M$ (see definition 8.11. That is, a redundant subdomain is such that if it is removed the partition still covers all labels. It is not our intent to discuss whether there are other definitions of redundant subdomains that are better, but merely to find an efficient way to either find a minimal partition as Meudec defines it or at least reduce the initial partition.

One way to reduce the partition size is to remove the redundant subdomains from the complete partition. Such techniques we refer to as being reduction-based.

Alternatively, we have rule-based techniques which we do not consider in this work. These techniques have the objective to reduce the size of
Chapter 9 Meudec’s Minimal Partition Method

the actual complete partition. For instance, in \texttt{vdmpart} [59], which is an implementation of Meudec’s partitioning strategy, the full-combination operator in some of the partition rules is replaced by a partial-combination operator. In this way the size of the complete partition is reduced. Of course, a further reduction to a minimal partition is possible, though this step is not performed in \texttt{vdmpart}.

9.1 Naive Approach

The naive approach, described by Meudec in his dissertation, works in the following way. Assume a partition on the form:

\[
\begin{align*}
&\left\{a_1, a_2\right\} \times \left\{a_3, a_4\right\} \times \left\{a_5, a_6\right\} \times \left\{a_7, a_8\right\} \\
&\times \left\{a_9, a_{10}\right\} \times \left\{b_1, b_2\right\} \times \left\{c_1, c_2\right\}
\end{align*}
\]

where only the labels of the atomic subdomains are shown, and where combinations \(a_1 \times a_3\), \(a_1 \times a_4\), \(a_6 \times a_7\), and \(a_6 \times a_0\) are unsatisfiable.

As partitioning proceed we continuously remove any inconsistent subdomains. For example:

\[
\left\{a_1, a_2\right\} \times \left\{a_3, a_4\right\} = \left\{a_{23}, a_{24}\right\}
\]

since both \(a_{13}\) and \(a_{14}\) are inconsistent. Similarly, we obtain

\[
\left\{a_5, a_6\right\} \times \left\{a_7, a_8\right\} = \left\{a_{57}, a_{58}\right\}
\]

After performing the distributed union the partition now is:

\[
\begin{align*}
&\left\{a_{23}, a_{24}\right\} \times \left\{a_9, a_{10}\right\} \times \left\{b_1, b_2\right\} \times \left\{c_1, c_2\right\}
\end{align*}
\]
performing the next combination gives us:

\[
\left\{ a_{239}, a_{023}, a_{249}, a_{024}, a_{579}, a_{057}, a_{589}, a_{058}, a_{689} \right\} \times \left\{ \begin{array}{c} b_1 \\ b_2 \end{array} \right\} \times \left\{ \begin{array}{c} c_1 \\ c_2 \end{array} \right\}
\]

The next combination to expand is between independent subdomains. Hence, as long as the labels are covered, any subset of the full combination is valid. Note that we do not have to check for consistency in this step, since the two subdomains are independent.

\[
\left\{ a_{239}, b_1, a_{023}, b_1, a_{249}, b_1, a_{024}, b_1, a_{579}, b_1, a_{057}, b_1, a_{589}, b_1, a_{058}, b_1, a_{689}, b_2 \right\} \times \left\{ c_1 \right\}
\]

Finally, after the last combination we obtain:

\[
\left\{ a_{239}, b_1, c_1, a_{023}, b_1, c_2, a_{249}, b_1, c_1, a_{024}, b_1, c_2, a_{579}, b_1, c_1, a_{057}, b_1, c_2, a_{589}, b_1, c_1, a_{058}, b_1, c_2, a_{689}, b_2, c_1 \right\}
\]

which is a minimal set of subdomains.

However, the naive approach has several drawbacks. One, for instance, is that it is too local in its scope. Because reduction of the inner subdomains is only performed locally, that is, reduction is performed only
when two subdomains are combined, problems will occur when two inner
subdomains on the same level share labels. For example, $P(a \lor b)$ results in

$$\begin{cases}
\{ a_1 \} \times \{ b_1 \} \\
\{ a_2 \} \times \{ b_1 \} \\
\{ a_2 \} \times \{ b_2 \}
\end{cases} \text{ naive} \rightarrow \begin{cases}
\{ a_1, b_1 \} \\
\{ a_2, b_1 \} \\
\{ a_2, b_2 \}
\end{cases}$$

Since the naive approach only performs local redundancy analysis, it will
not realize that a minimal partition in this situation is:

$$\begin{cases}
\{ a_1, b_1 \} \\
\{ a_2, b_2 \}
\end{cases}$$

### 9.2 Graph Approach

In an attempt to solve the first situation with the naive approach, where
the order of combination of the different subdomains resulted in an in-
adquate partition, Meudec suggested that a minimal set of subdomains
should be derived directly from a graph corresponding to the partition.
For example, take the following partition and its corresponding graph in
figure 9.1.

$$\begin{cases}
\{ a_1 \} \times \{ b_1 \} \\
\{ a_2 \} \times \{ b_1 \} \\
\{ a_2 \} \times \{ b_2 \} \\
\{ c_1 \} \times \{ c_2 \}
\end{cases}$$

Meudec points out that a minimal set subdomains for this partition is
equivalent to a minimal set of paths that together cover all nodes in the
tagraph.

Still, we have not handled the situation where partitions contain de-
pendent labels. For example, assume a dependent partition of the form:

$$\begin{cases}
\{ a_1 \} \\
\{ a_2 \}
\end{cases} \times \begin{cases}
\{ b_1 \} \\
\{ b_2 \}
\end{cases} \times \begin{cases}
\{ a_3 \} \\
\{ a_4 \}
\end{cases}$$

with its partition graph in figure 9.2. In this situation it is not sufficient
to find a minimal set of paths that covers all nodes, since this would leave
9.2 Graph Approach

Figure 9.2: A dependent partition graph.

Figure 9.3: Merging the nodes of the graph in figure 9.2 gives an independent graph.

out the necessary combinations of the dependent nodes. In our case, some paths contain dependent nodes, for instance, the path $a_1, b_2, a_3$ contains the dependent nodes $a_1$ and $a_3$. Therefore, in order to come up with a minimal partition we must not only cover all labels, but also cover all combinations of dependent labels.

Meudec suggests that the graph would be transformed, somehow, such that dependent nodes are not of concern. Specifically, he proposes that nodes in the same path are made adjacent and merged (figure 9.3). Consequently, no path will contain dependent nodes. Thus, the problem of finding a minimal partition is reduced to:

1. Merge all dependent vertices.
2. Find a minimal path cover.

However, for more complex graphs Meudec found the merging process very difficult. To illustrate the complexity of the graph transformation problem he gives the following example where the $d_i$:s are dependent:

\[
\begin{align*}
\{a_1\} \times \left\{ \begin{array}{c} b_1 \\ b_2 \\ b_3 \\ b_4 \end{array} \right\} & \times \left\{ \begin{array}{c} c_1 \\ c_2 \end{array} \right\} \times \left\{ \begin{array}{c} d_1 \\ d_2 \end{array} \right\} \\
\{a_2\} 
\end{align*}
\]

Figure 9.4 and figure 9.5 corresponds to the expression’s unmerged and merged graphs. Failing to find a systematic solution of the vertex merging problem, the graph approach was abandoned. He does not specify how to derive a minimal path cover either. He suggests that a deeper study of graph theory may be fruitful, especially bipartite graphs and set covers.
Figure 9.4: A complex graph.

Figure 9.5: The graph in figure 9.4 with merged nodes.
9.3 Analysis of Meudec’s Minimal Partition Algorithm

Meudec concludes that devising an efficient algorithm for finding a minimal partition is difficult. In his final approach he aims at a more straightforward solution. In short, it is close to the naive approach in that the partition skeleton is expanded and inconsistent expressions are removed along the way. However, combinations are always fully expanded, that is, no other reduction is performed. In a second step, once the complete partition is derived a minimal one is sought.

This approach has the advantage that combinations are not ruled out too early because of local scope, as in the naive approach. However, it is inefficient since the complete partition must first be fully expanded and then minimized. Below we present the algorithm as given by Meudec.

He points out that there may be many minimal partitions and that it is therefore non-deterministic.

```plaintext
// initial call: minimal(expand_F(P(e)), ∅).
function minimal(P, M : set of subdomains) : set of subdomains
    if P = ∅ then return M
    if ∃s ∈ P : ∃s′ ∈ P ∪ M : labels(s) ⊆ labels(s′) then
        return minimal(P − {s}, M)
    if ∃s ∈ P : ∃l ∈ labels(s) : ¬ (∃s′ ∈ P ∪ M : l ∈ labels(s′))
        return minimal(P − {s}, M ∪ {s})
    let s ∈ P in
        return minimal(P − {s}, M)
end minimal
```

The function takes two arguments $P$ and $M$, where the first is the complete (consistent) partition and the second is a variable used for building the minimal partition, which is later returned. The body of the function consists of four cases that either return directly or recursively call minimal:

1. **(Minimal)** if $P$ is equal to the empty set then all subdomains have been considered. Return $M$.

2. **(Subsumed)** if there is a subdomain $s$ that is subsumed ($⊆$) by some other subdomain, then $s$ should not be part of any minimal subdomain.

3. **(Uncovered)** if there is a subdomain $s$ with a label, represented in neither $P$ nor $M$, then $s$ must be part of the minimal partition.
Chapter 9 Meudec’s Minimal Partition Method

4. *(Redundant)* if none of the above is true, remove an arbitrary subdomain from \( P \).

1 Note that \( P \cup M \) must be interpreted as \( (P - \{s\}) \cup M \) for this to work.

**Observation 9.1.** If case 2 holds then \( M = \emptyset \).

*Proof.* Assume that case 2 holds on recursion \( j \) and that \( M \neq \emptyset \). This implies that case 3 must have been true on some earlier recursion \( i \), since case 3 is the only point where new elements in \( M \) are introduced. However, for case 3 to be executed on recursion \( i \) case 2 must have been false, meaning that there were no subsumed subdomains. In turn, this means that for case 2 to hold on recursion \( j \) new elements must have been introduced in \( P \), but this is indeed impossible.

As a result of observation 9.1 one can factor out case 2 from minimal, and instead remove any subsumed subdomains before deriving a minimal partition.

A sample run of minimal on partition (8.16) is given in table 9.2(a). The table should be read as follows. *Add* and *rem* corresponds to case 3 and 4, respectively. On recursion 1 only case 4 matches, therefore an arbitrary subdomain can be removed. Assume that \( s_2 \) is removed. On rec. 2 the removal of \( s_2 \) made \( s_4 \) unique, since it is the only one left covering label \( x_2 \). Therefore, \( s_4 \) is added to \( M \). Continuing in this manner minimal will in the 12th recursion return a set containing the following subdomains \{ \( s_1, s_4, s_9, s_{12} \) \}, which is known to be minimal (see previous section).

Roughly, the algorithm has a complexity of at most \( O(n^3) \), where \( n \) is the number of subdomains in \( P \). As the problem of finding a minimal partition is very reminiscent of a set cover problem, which are known to be exponential, it is unlikely that this algorithm really works. In fact, the counterexample given in table 9.2(b) confirms this doubt. Here minimal returns the partition \{ \( s_3, s_4, s_5, s_8, s_9, s_{11} \) \}, which is not minimal. In this alternative run, a weakness in case 4 has been exploited. In case 4 it is permissible to remove any remaining subdomain from \( P \). By consistently removing the worst possible subdomain from \( P \), seen from a minimalistic point of view, the algorithm is forced to include more subdomains than necessary in \( M \).

**Theorem 9.1.** Meudec’s algorithm (minimal) does not always return minimal partitions.

*Proof.* See table 9.2(b).
### 9.3 Analysis of Meudec’s Minimal Partition Algorithm

Table 9.1: Two different executions of function minimal on the same complete partition.

<table>
<thead>
<tr>
<th>Rec.</th>
<th>Action</th>
<th>Subd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rem</td>
<td>$s_2$</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>$s_4$</td>
</tr>
<tr>
<td>3</td>
<td>rem</td>
<td>$s_3$</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>$s_1$</td>
</tr>
<tr>
<td>5</td>
<td>rem</td>
<td>$s_5$</td>
</tr>
<tr>
<td>6</td>
<td>rem</td>
<td>$s_6$</td>
</tr>
<tr>
<td>7</td>
<td>rem</td>
<td>$s_7$</td>
</tr>
<tr>
<td>8</td>
<td>rem</td>
<td>$s_8$</td>
</tr>
<tr>
<td>9</td>
<td>rem</td>
<td>$s_{10}$</td>
</tr>
<tr>
<td>10</td>
<td>add</td>
<td>$s_9$</td>
</tr>
<tr>
<td>11</td>
<td>rem</td>
<td>$s_{11}$</td>
</tr>
<tr>
<td>12</td>
<td>add</td>
<td>$s_{12}$</td>
</tr>
</tbody>
</table>

(a) Minimal

<table>
<thead>
<tr>
<th>Rec.</th>
<th>Action</th>
<th>Subd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rem</td>
<td>$s_{12}$</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>$s_{11}$</td>
</tr>
<tr>
<td>3</td>
<td>rem</td>
<td>$s_{10}$</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>$s_9$</td>
</tr>
<tr>
<td>5</td>
<td>rem</td>
<td>$s_6$</td>
</tr>
<tr>
<td>6</td>
<td>add</td>
<td>$s_8$</td>
</tr>
<tr>
<td>7</td>
<td>rem</td>
<td>$s_1$</td>
</tr>
<tr>
<td>8</td>
<td>add</td>
<td>$s_3$</td>
</tr>
<tr>
<td>9</td>
<td>rem</td>
<td>$s_2$</td>
</tr>
<tr>
<td>10</td>
<td>add</td>
<td>$s_4$</td>
</tr>
<tr>
<td>11</td>
<td>add</td>
<td>$s_5$</td>
</tr>
<tr>
<td>12</td>
<td>rem</td>
<td>$s_7$</td>
</tr>
</tbody>
</table>

(b) Not minimal
Chapter 10

Tree-based Partial Expansion

— Frequently, crashes are followed with a message like 'ID 02'. 'ID' is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had.

Guy Kawasaki

As we have learned in chapter 9, Meudec pointed out that an important step in finding a minimal partition is to merge all dependent vertices of the partition graph. He suggested that the dependent nodes in the partition graph should be made adjacent and then merged. The resulting graph is called a dependence-free partition graph because it contains no paths with dependent nodes.

However, because of the complexity of this approach it was abandoned. Despite the difficulty of the problem, it is challenging to find a general solution. Not only will this help us in finding a way to find a minimal partition, but it also gives us a way to remove inconsistent subdomains from a partition. In dependence-free partitions each node can be solved independently of the others, therefore, if a node is inconsistent it can be removed from the graph since all paths (subdomains) through that node are inconsistent.

In this chapter a solution to the vertex merging problem is presented. Instead of manipulating the partition in graph form, trees are used.
Chapter 10  Tree-based Partial Expansion

10.1 Vertex Merging

Let us go back to the partition given in (9.1). The partition can be stated as an ordinary arithmetic expression.

\[
(a_1 + a_2)((b_1 + b_2)(c_1 + c_2) + (b_3 + b_4)(d_1 + d_2)) \\
\cdot ((e_1 + e_2)(d_3 + d_4) + (e_3 + e_4)(f_1 + f_2))
\]

Let \(X_{ij}\) be an abbreviation for \(x_i + x_j\). Normally, when deriving the fully expanded partition, we would multiply all terms together such that we are left with only a set of atomic subdomains. Now, instead of expanding everything in our way, we will adopt a more conservative approach: partial expansion. Using standard arithmetic laws, we will carefully manipulate the partition such that as little as necessary is expanded. As a matter of fact, later in this chapter we will prove that the result is a partition graph where all dependent nodes are merged.

Consider the expression below. In the normal partitioning process we would combine subdomains 1 and 2. However, as the two domains do not share dependent labels this is not necessary.

\[
A_{12} (B_{12}C_{12} + B_{34}D_{12}) (E_{12}D_{34} + E_{34}F_{12})
\]

Remember that in the graph approach, only a graph composed of dependent node labels in the same path needed to be handled specially. A subgraph of only subdomains 1 and 2 does not contain dependent labels, and, therefore, it is already dependence-free.

In this particular partition, though, subdomains 2 and 3 share dependent labels. Since they are combined with \(\times\) we know that they share a path in the partition graph. If two dependent domains are expanded (combined), it will either result in a new label if the two are atomic domains, or it will result in some new combinations of subdomains where some or all share dependencies. Here, a combination of 2 and 3 leads to four new subdomains.

\[
A_{12}((B_{12}C_{12}) (E_{12}D_{34} + E_{34}F_{12}) + (B_{34}D_{12}) (E_{12}D_{34} + E_{34}F_{12}))
\]

Subdomains 3 and 4 are the only subdomains that are combined and share dependencies, and thus, those are the only ones that need to be expanded. However, subdomain 3 is composed of two combined subdomains, where
10.2 Proof of Concept

$D_{12}$ is the depending one. In order to expand as little as necessary the expression is rearranged.

$$A_{12}((B_{12}C_{12})(E_{12}D_{34} + E_{34}F_{12}) + B_{34} \Bigg[ D_{12} \left( E_{12}D_{34} + E_{34}F_{12} \right) \Bigg])$$

The result of the combination of 1 and 2 is:

$$A_{12}((B_{12}C_{12})(E_{12}D_{34} + E_{34}F_{12}) + B_{34}(D_{12}D_{34}E_{12} + D_{12}E_{34}F_{12}))$$

At this point only subdomains 1 and 2 are dependent. After expanding 1 and 2 we obtain:

$$A_{12}((B_{12}C_{12})(E_{12}D_{34} + E_{34}F_{12}) + B_{34}(D_{12}D_{34}E_{12} + D_{12}E_{34}F_{12}))$$

where $D_{12} = D_{12} \times D_{34} = \{d_{13}, d_{14}, d_{23}, d_{24}\}$. Since the remaining subdomains are atomic, no further expansion is needed. It is easy to see that this process corresponds to a partial expansion of the partition. Below we show the partially expanded partition.

\[
\begin{align*}
\{a_1, a_2\} \times \left( \begin{array}{c}
\{b_1\} \times \{c_1\} \\
\{b_2\} \times \{c_2\}
\end{array} \right) \\
\left( \begin{array}{c}
\{d_{13}\} \times \{e_1\} \\
\{d_{14}\} \times \{e_1\} \\
\{d_{23}\} \times \{e_1\} \\
\{d_{24}\} \times \{e_1\}
\end{array} \right) \\
\left( \begin{array}{c}
\{f_1\} \\
\{f_2\}
\end{array} \right)
\end{align*}
\]

10.2 Proof of Concept

**Definition 10.1.** Let $\text{nodes}(S)$ be a function that returns the set of labels of all atomic subdomains in $S$.

$$\text{nodes}(S) = \begin{cases} 
\text{labels}(S) & \text{if } S \text{ is atomic} \\
\text{nodes}(S_1) \cup \text{nodes}(S_2) & \text{if } S = S_1 \times S_2 \\
\text{nodes}(s) \cup \text{nodes}(S - \{s\}) & \text{otherwise, for some } s \in S
\end{cases}$$

In contrast to $\text{labels}(S)$ which returns union of all labels of all atomic subdomains in a fully expanded $S$, $\text{nodes}(S)$ returns that union of labels
Chapter 10 Tree-based Partial Expansion

that are currently represented in $S$ (without expanding $S$). Consequently, $\text{nodes}(S) = \text{labels}(S)$ when $S$ is fully expanded. For example, below are the nodes of partition (9.1) and (10.1).

$\text{nodes}(9.1) = \{a_1, a_2, b_1, b_2, b_3, b_4, c_1, c_2, d_1, d_2, d_3, d_4, e_1, e_2, e_3, e_4, f_1, f_2\}$

$\text{nodes}(10.1) = \text{nodes}(9.1) \cup \{d_{13}, d_{14}, d_{23}, d_{24}\}$

**Definition 10.2.** Two domains $A$ and $B$ are dependent, $\text{dep}(A, B)$, if and only if

$$\exists a \in A : \exists b \in B : \text{dep}(a, b)$$

**Definition 10.3.** A domain $S = s_1 \times s_2$ is called *dependence free* if $s_1$ and $s_2$ are independent *and* dependence free. A domain $S = \{s_1, \ldots, s_n\}$ is dependence free if all $s_i$ are dependence free. An atomic domain is always dependence free.

That is, for a subdomain to be dependence free it must not be (recursively) composed of products of dependent subdomains.

**Theorem 10.1.** A domain $S$ is dependence free if and only if

$$\text{nodes}(S) = \text{labels}(S).$$

*Proof.* ($\Rightarrow$) Since $S$ is dependence free it cannot be composed of products of dependent subdomains and since new labels are only created when two dependent atomic domains are combined (see definition 8.6), no more labels than those currently in $S$ will be created in a full expansion of $S$. Thus, $\text{nodes}(S) = \text{labels}(S)$ must hold.

($\Leftarrow$) Assume $\text{nodes}(S) = \text{labels}(S)$ holds. This means that no new labels have been created when expanding $S$, that is, there has been no combination of dependent atomic domains. Since only products of dependent atomic domains introduces labels, $S$ cannot have any. By definition $S$ is dependence free.

This implies that the solution of the vertex merging problem is to find a singleton, dependence-free partition graph. A partition graph being singleton means that its nodes only consists of one label each. That is, the corresponding partition of the graph consists only of atomic subdomains with one label.

**Definition 10.4.** A singleton (atomic) domain is an atomic domain consisting of only one label. Note that all terminal domains are singleton, but necessarily the other way around.

A compound domain being singleton means that all of its atomic subdomains are singleton.
10.3 The Tree Rules

The partial expansion algorithm given here is based on a set of 10 rewrite rules that operate on the abstract syntax tree of the arithmetic representation of the partition. Of course, a more efficient approach would be to integrate the process into the actual partitioning. However, to simplify the discussion it is put in this way.

A partition can be modeled using the standard arithmetic operators for addition and multiplication. The set notation \( \{s_1, \ldots, s_n\} \) amounts to addition \( s_1 + \cdots + s_n \) and the full combination operator \( \times \) amounts to multiplication. It is easy to confirm that the following properties are true:

1. **(Commutative laws)** \( a + b = b + a \) and \( a \cdot b = b \cdot a \).

2. **(Associative laws)** \( (a + b) + c = a + (b + c) \) and \( (a \cdot b) \cdot c = a \cdot (b \cdot c) \).

3. **(Distributive law)** \( (a + b) \cdot c = a \cdot c + b \cdot c \).

The algorithm, called expand\(_P\), starts out with an unexpanded partition and recursively walks its subdomains and modifies them according to the arithmetic laws given above. Partial expansion is less greedy than full expansion, since only dependent subdomains are expanded; others are left as is. By induction, the resulting partially expanded partition must be singleton and dependence free.

The predicate \( \text{dep}(t_1, t_2) \) is true iff the two trees are dependent. The relation is not transitive. The left and right subtree of the top node is referred to as \( l \) and \( r \) respectively. Addition nodes are denoted by \( + \), multiplication nodes are denoted by \( \cdot \). Atoms, that is, (singleton) atomic subdomains, are denoted by \( a \). Arbitrary trees are denoted by \( t_i \). A circled node in the right hand side of a rewrite rule indicates where expansion should continue (a recursive call to expand\(_P\)). In case of several expansion points, the order of continuation is left to right and bottom to top. The 10 rewrite rules as well as the function expand\(_P\) given below are followed by a detailed rationale.

\[
\begin{align*}
R1: & \quad a \rightarrow a \\
R2: & \quad \begin{array}{c}
a_1 \\
\downarrow \quad \text{dep}(l,r)
\end{array} \rightarrow \begin{array}{c}
a_2 \\
\longrightarrow \quad \text{merge}(a_1, a_2)
\end{array}
\end{align*}
\]
Chapter 10 Tree-based Partial Expansion

(R3)

(R4)

(R5)

(R6)

(R7)
10.4 Rationale

The rewrite rules have been designed to cover all possible combinations of trees. That is, all trees from a single atom up to \((t_1 + t_2) \cdot (t_3 + t_4)\).

The main idea is to consider the best possible rewrite for the current tree given the following basic rewrites:

1. (Identity) \(t \rightarrow t\)
2. (Creative) \(a_1 \cdot a_2 \rightarrow \text{merge}(a_1, a_2)\)
3. (Distributive) \(t_1 \cdot (t_2 + t_3) \rightarrow t_1 \cdot t_2 + t_1 \cdot t_3\)

\[
\begin{align*}
\text{R8} & : a/+ \quad \text{dep}(l, t_1) \quad t_2 \\
\text{R9} & : a/+ \quad \text{dep}(l, t_2) \quad t_1 \\
\text{R10} & : a/+ \\
\end{align*}
\]

\[
\begin{align*}
\text{function} \ & \text{expand}_p(t : \text{tree}) : \text{tree} \\
\text{for} \ & i = 1 \ \text{to} \ \text{length}(\text{rules}) \\
\ & \text{if} \ (\text{match}(\text{rules}[i], \ t)) \ \text{then} \\
\ & \quad \text{return} \ \text{apply}(\text{rules}[i], \ t); \\
\text{end for} \\
\ & // \ \text{Will never get here.} \\
\text{end expand}_p
\end{align*}
\]
4. (Associative) \((t_1 \cdot t_2) \cdot t_3 \longleftrightarrow t_1 \cdot (t_2 \cdot t_3)\)

5. (Commutative) \(t_1 \cdot t_2 \longleftrightarrow t_2 \cdot t_1\)

**Definition 10.5.** Let \(T\) be a partition tree and denote its domain by \(S_T\). Then a rewrite rule \(r\) is called **domain preserving** if an application of \(r\) on a subtree of \(T\) results in a new tree \(T'\) such that \(S_{T'} = S_T\).

In other words, a rewrite rule not being domain preserving means that when a applied on a subtree it adds or removes subdomains from the domain of the supertree, and therefore, changes the meaning of the original domain. Naturally, proper expansion rules must not behave in this way. However, it is fairly straightforward to show that the basic rewrites above indeed are domain preserving. Thus transformations of partition trees based on these rewrites are also domain preserving.

As can be seen from the basic rewrites above, trees are never unexpanded into a possibly more compact form. Using unexpansion would possibly lead to more compact partitions, however, such an algorithm is probably more difficult to devise. Instead, the algorithm tries its best to rearrange (commutative and associative rewrites) or expand (creative or distributive rewrites) the given expression. Special care must be taken in order to not end up in a loop of associative or commutative rewrites. This could happen if the same tree is reached via a series of rewrites. In situations where neither rearrangement nor expansion is appropriate a fall-back to the identity rewrite can be used.

**(R1) – Identity** If a single atom is encountered no further expansion is possible.

**(R2) – Creative** This rule reflects the situation where two atomic subdomains are about to be combined. If the two atoms are dependent then, by definition of \(\times\), a new label will be created. Therefore, the tree should be replaced by the new label. However, if it is not the case that the two atoms are dependent this rule should not be applied. Instead the incoming tree can be returned. This is done indirectly, since the incoming tree will match rule (R10).

**(R3) – Distributive** Consider the following situation \(a(t_1 + t_2)\). If the atom is dependent on any of the two subtrees then an application of the distributive rewrite is appropriate, because the atom will come closer a merge situation (R2). If the atom is not dependent on the two trees it is safe to apply rule (R10) instead.
For example, assume the incoming expression $a_1(b_1 + b_2(c_1 + a_2))$. Since the left subtree of the root is an atom and dependent on the right subtree (because of the $a_2$ label) rule (R3) is applied.

![Diagram of tree with labels $a_1$, $b_1$, $b_2$, $c_1$, $a_2$ and transformation with rule (R3) indicated by arrow]

The resulting tree corresponds to the following expression $a_1 b_1 + a_1 b_2 (c_1 + a_2)$. See how the dependent $a_1$ has come closer to $a_2$. Note that partial expansion is continued along the two subtrees of the new root. Thus, dependent labels successively move closer to a merge.

(R4) – Commutative The idea of commutative rules is to rewrite the expression in a form such that it can be handled by some of the other rules. Doing this simplifies discussion. For reasons explained in the rationale for rule (R5), left and right subtrees of the root need to be swapped if $t_1$ and $t_2$ are dependent and if they, in turn, are dependent on either (but not both) $t_3$ and $t_4$.

(R5) – Distributive Rule (R4) and rule (R5) both match the same tree structure, but require different dependencies. Consider an expression $(t_1 + t_2)(t_3 + t_4)$, matching the structure of the two rules. Assuming that the left and right subtrees of the root are dependent, one might think that a suitable action would be to expand the root. That is, the following transformation is performed.

$$(t_1 + t_2)(t_3 + t_4) = t_1 t_3 + t_1 t_4 + t_2 t_3 + t_2 t_4$$

This would be fine in a situation where all the $t_i$:s are of the same dependence class. For example:

$$(a_1 + a_2)(a_3 + a_4) = a_1 \cdot a_3 + a_1 \cdot a_4 + a_2 \cdot a_3 + a_2 \cdot a_4$$
However, if only portions of the left and right subtrees are dependent, then the above strategy is not well suited, since the expression is expanded more than needed.

\[(b_1 + a_1)(a_2 + a_3) = b_1 \cdot a_2 + b_1 \cdot a_3 + a_1 \cdot a_2 + a_1 \cdot a_3\]

A better transformation would be to keep \((a_2 + a_3)\) intact as long as it is not absolutely necessary to expand it. Thus, using the distributive rewrite gives an expression that is less expanded.

\[(b_1 + a_1)(a_2 + a_3) = b_1(a_2 + a_3) + a_1(a_2 + a_3)\]

If partial expansion is continued on the two subexpressions \(b_1(a_2+a_3)\) and \(a_1(a_2+a_3)\) only the latter will be expanded and therefore the resulting expression is more compact.

\[b_1(a_2 + a_3) + a_1(a_2 + a_3) = b_1(a_2 + a_3) + a_1 a_2 + a_1 a_3\]

Hence, in the situation of \((t_1 + t_2)(t_3 + t_4)\) it is not sufficient to only match dependency between left and right subtrees, but dependency must also be matched among the \(t_i\):s. Table 10.1 lists all situations where left and right subtrees of the root node are dependent. For instance, situation 3 corresponds to an expression where \(t_1\) belongs to dependence class \(b\) and \(t_2\), \(t_3\), and \(t_4\) belong to dependence class \(a\). Thus, the expression \((a_1 + b_1) \cdot (a_2 + a_3)\) given above matches situation 2.

It is easy to reason that situations 1–7 can all be treated by the distributive rewrite in rule (R5). Situation 8 and 9, on the other hand, are handled by a commutative rewrite in rule (R4). Then, in the next recursion, rule (R5) distributes the subtrees. Furthermore, from table 10.1 it can be seen that it is impossible to end up in a non-terminating loop over rule (R4).

Note that the dependence condition in (R5) is simplified to \(\text{dep}(l, r)\). This is possible since the partial expansion algorithm always handles the rules in sequential order. The actual dependence condition of rule (R5) is given below.

\[\text{dep}(l, r) \land \neg(\text{dep}(t_1, t_2) \land \text{dep}(t_1, t_3) \lor \text{dep}(t_1, t_4))\]

\(\text{negated dependence condition of rule (R4)}\)

(R6) – Commutative As mentioned earlier, to keep discussion simple, some expressions are rewritten with the commutative rewrite. This rule swaps the two subtrees and relies on rule (R3) for further expansion.
Table 10.1: All situations where left and right subtrees are dependent for the expression \((t_1 + t_2) \cdot (t_3 + t_4)\), where \(a\) is a dependence class and \(*\) is a wildcard class other than \(a\).

<table>
<thead>
<tr>
<th></th>
<th>(t_1)</th>
<th>(t_2)</th>
<th>(t_3)</th>
<th>(t_4)</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(R5)</td>
</tr>
<tr>
<td>2</td>
<td>(a)</td>
<td>(*)</td>
<td>(a)</td>
<td>(a)</td>
<td>(R5)</td>
</tr>
<tr>
<td>3</td>
<td>(*)</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(R5)</td>
</tr>
<tr>
<td>4</td>
<td>(a)</td>
<td>(*)</td>
<td>(a)</td>
<td>(*)</td>
<td>(R5)</td>
</tr>
<tr>
<td>5</td>
<td>(a)</td>
<td>(*)</td>
<td>(*)</td>
<td>(a)</td>
<td>(R5)</td>
</tr>
<tr>
<td>6</td>
<td>(*)</td>
<td>(a)</td>
<td>(a)</td>
<td>(*)</td>
<td>(R5)</td>
</tr>
<tr>
<td>7</td>
<td>(*)</td>
<td>(a)</td>
<td>(*)</td>
<td>(a)</td>
<td>(R5)</td>
</tr>
<tr>
<td>8</td>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(*)</td>
<td>(R4)</td>
</tr>
<tr>
<td>9</td>
<td>(a)</td>
<td>(a)</td>
<td>(*)</td>
<td>(a)</td>
<td>(R4)</td>
</tr>
</tbody>
</table>

(R7) – Right Associative  This rule, which matches the expression \((t_1 \cdot t_2) \cdot t_3\), handles the situation where a possible candidate for expansion is buried in the left subtree \(t_1 \cdot t_2\). The associative law is applied to make \(t_1\) the left child of the root: \(t_1 \cdot (t_2 \cdot t_3)\). Partial expansion is continued from the root and in the next recursion either of the following rules will be applied: (R8), (R9), or (R10).

(R8) – Left Associative  At a first glance it might seem that this rule together with rule (R7) may contribute to a non-terminating loop, since structurally, they are inverses of each other. However, since rule (R8) continues its partial expansion on the left child before partially expanding the root, the root will never match rule (R7) with the same left child. Since the two children of the left subtree of the root are dependent, the left subtree will eventually be rewritten by some of the other rules. Therefore, if the root node matches (R7) in some later rewrite the subtrees will be different. As an example consider the partial expansion of \((a_1 \cdot a_2) \cdot (a_3 + b_1)\).
The initial tree is rewritten with rule (R7) and recursion is continued on the root. This rewrite corresponds to \((a_1 \cdot a_2) \cdot (a_3 + b_1) = a_1 \cdot (a_2 \cdot (a_3 + b_1))\), that is, associativity is changed from left to right. Next, rule (R8) is applied. Note how associativity is switched back to left again, however, this time recursion is set to first continue on the left child and then on the root.

Therefore, before the root is matched with rule (R7), the left child is first expanded by rule (R2).

Rule (R2) merges the two labels into one, thus when root node is considered it does not match rule (R7) any more, but rule (R3) instead.

(R9) – Commutative If the left child is dependent on \(t_2\) then \(t_1\) and \(t_2\) swapped. Doing this enables rule (R8) to handle the partial expansion in the following recursion.
(R10) – Default  If none of the rewrites match, then this rule defines the default behavior. Partial expansion is continued on the left and right child.

The list of table 10.2 shows all possible combinations of incoming trees. From the list it can be seen that all combinations are handled by at least one rule. In the case multiple rules are listed only one matches because of the dependence condition.

Table 10.2: A complete list of trees considered. If no dependence then default to rule (R10).

<table>
<thead>
<tr>
<th></th>
<th>Root</th>
<th>Left</th>
<th>Right</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>-</td>
<td>-</td>
<td>(R1)</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>a</td>
<td>a</td>
<td>(R10)</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>+</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>·</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>a</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>·</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>8</td>
<td>·</td>
<td>a</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>9</td>
<td>·</td>
<td>+</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>10</td>
<td>·</td>
<td>·</td>
<td></td>
<td>(R10)</td>
</tr>
<tr>
<td>11</td>
<td>·</td>
<td>a</td>
<td>a</td>
<td>(R2)</td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>+</td>
<td></td>
<td>(R3)</td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>·</td>
<td></td>
<td>(R8)  or (R9)</td>
</tr>
<tr>
<td>14</td>
<td>+</td>
<td>a</td>
<td></td>
<td>(R6)</td>
</tr>
<tr>
<td>15</td>
<td>+</td>
<td>+</td>
<td></td>
<td>(R4)  or (R5)</td>
</tr>
<tr>
<td>16</td>
<td>+</td>
<td>·</td>
<td></td>
<td>(R8)  or (R9)</td>
</tr>
<tr>
<td>17</td>
<td>·</td>
<td>a</td>
<td></td>
<td>(R7)</td>
</tr>
<tr>
<td>18</td>
<td>·</td>
<td>+</td>
<td></td>
<td>(R7)</td>
</tr>
<tr>
<td>19</td>
<td>·</td>
<td>·</td>
<td></td>
<td>(R7)</td>
</tr>
</tbody>
</table>
Chapter 11
Set-cover Method

— The test of an outfielder’s skill comes when he has to go against the fence to make a catch.

Joe DiMaggio

We have learned that finding a minimal partition is not a simple task. In table 11.1 we see that the number of subdomains grows with $\Theta(2^n)$ for the expression $P(a_1 \lor \cdots \lor a_n)$. Matters get even worse if we assume $P(\exists: a_1 \lor \cdots \lor a_n)$. As seen in table 11.2 the growth is $\Theta(2^{2^n})$. Clearly, it is not very efficient to first generate all subdomains, and then in a later stage discard a large portion of them.

However, at this point we do not have a working algorithm. Therefore, it is important to find out if the problem has an algorithmic solution. Perhaps we must settle for an algorithm that only finds approximately minimal partitions, or maybe we have to modify the partitioning rules such that they are more efficient in terms of generating fewer redundant subdomains. Finding a general solution, even though inefficient, to the minimal partition problem may help in deriving better algorithms or approximate ones.

Table 11.1: Subdomain growth for $P(a_1 \lor \cdots \lor a_n)$, with $n > 1$.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Subdomains</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1 \lor a_2$</td>
<td>$2^2 - 1 = 3$</td>
</tr>
<tr>
<td>$a_1 \lor a_2 \lor a_3$</td>
<td>$2^3 - 1 = 7$</td>
</tr>
<tr>
<td>$a_1 \lor a_2 \lor a_3 \lor a_4$</td>
<td>$2^4 - 1 = 15$</td>
</tr>
<tr>
<td>$a_1 \lor a_2 \lor a_3 \lor a_4 \lor a_5$</td>
<td>$2^5 - 1 = 31$</td>
</tr>
<tr>
<td>$a_1 \lor \cdots \lor a_n$</td>
<td>$2^n - 1$</td>
</tr>
</tbody>
</table>
Table 11.2: Subdomain growth for $P(\exists : a_1 \lor \ldots \lor a_n)$, with $n > 1$.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Subdomains</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists : a_1 \lor a_2$</td>
<td>$2^4 + 1 = 17$</td>
</tr>
<tr>
<td>$\exists : a_1 \lor a_2 \lor a_3$</td>
<td>$2^8 + 1 = 257$</td>
</tr>
<tr>
<td>$\exists : a_1 \lor a_2 \lor a_3 \lor a_4$</td>
<td>$2^{16} + 1 = 65535$</td>
</tr>
<tr>
<td>$\exists : a_1 \lor a_2 \lor a_3 \lor a_4 \lor a_5$</td>
<td>$2^{32} + 1 = 4294967297$</td>
</tr>
<tr>
<td>$\exists : a_1 \lor \ldots \lor a_n$</td>
<td>$2^{2^n} + 1$</td>
</tr>
</tbody>
</table>

### 11.1 Set Covers

One might define optimization as the science of finding the best solution to certain mathematically defined problems. This science covers a wide area of research, for instance, the study of optimality criteria for problems and finding fast algorithms for solving such problems. Before 1940 there was little known in this area, but in the advent of the computer the development of optimization methods took off. [46]

In this approach for finding a minimal partition, called the set-cover method, Meudec’s problem definition of a minimal partition is transformed to an integer programming (IP) problem known as a set cover. This is advantageous as there are well-known algorithms to use in order to solve such problems, for example using the Simplex method and branch-and-bound (see section 5.2).

A partition $P$ can be given in matrix form as an $m \times n$ matrix $A$, where the $m$ rows correspond to the labels and the $n$ columns to each subdomain’s set of labels in the fully expanded partition. Let $a_{ij}$ be the value in row $i$ and column $j$. Let $a_{ij} = 0$ mean that the $i$:th label is not represented in the $j$:th subdomain and let $a_{ij} = 1$ mean that it is. Below is partition (8.16) in matrix form.

$$
A = \begin{pmatrix}
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{pmatrix}
$$

$x_1$

$x_2$

$x_3$

$x_4$

$y_1$

$y_2$

$y_3$

$y_4$

The columns of $A$ correspond to the labels of the subdomains. The labels of the fifth subdomain $\{x_1, y_3\}$, for example, correspond to the fifth
11.1 Set Covers

column $A_5 = (1, 0, 0, 0, 0, 0, 1, 0)^T$, where 0 means that the subdomain
does not contain the corresponding label, while 1 means that it does.

Now, assume that zeros are transparent and that the ones are not.
Imagine now that you are peeking through one of the rows of $A$ from the
left side. Clearly, if the label on the other side of the row is visible that
means that row contains only zeros. If the label is not visible then at
least one element is 1. Thus, our objective is to find the smallest set of
columns of $A$ such that the right side is completely invisible.

To do this, assume that the sum of row 8 is calculated, that is, the sum
of the row representing label $y_4$. The sum of a row is equal to the number
of times a certain label is represented in the partition. In this case $y_4$ is
represented 4 times.

$$
\sum_{j=1}^{n} a_{8,j} = a_{8,1} + a_{8,2} + \cdots + a_{8,n} = 0 + 0 + \cdots + 1 = 4
$$

While each row corresponds to a label, each column corresponds to a
subdomain. Thus, calculating the row sum only for a selected number of
columns would be a way to determine whether the selected columns (sub-
domains) can represent a minimal partition. For example, given below is
the row sum of the 8 labels for column 1, 2, and 7.

\[
\begin{align*}
x_1 &: a_{1,1} + a_{1,2} + a_{1,7} = 1 + 1 + 0 = 2 \\
x_2 &: a_{2,1} + a_{2,2} + a_{2,7} = 0 + 0 + 1 = 1 \\
x_3 &: a_{3,1} + a_{3,2} + a_{3,7} = 0 + 0 + 0 = 0 \\
x_4 &: a_{4,1} + a_{4,2} + a_{4,7} = 0 + 0 + 0 = 0 \\
y_1 &: a_{5,1} + a_{5,2} + a_{5,7} = 1 + 0 + 0 = 1 \\
y_2 &: a_{6,1} + a_{6,2} + a_{6,7} = 0 + 1 + 0 = 1 \\
y_3 &: a_{7,1} + a_{7,2} + a_{7,7} = 0 + 0 + 1 = 1 \\
y_4 &: a_{8,1} + a_{8,2} + a_{8,7} = 0 + 0 + 0 = 0
\end{align*}
\]

It is easy to see that this cannot be a minimal partition since $x_3$, $x_4$, and
$y_4$ are not represented, that is, all labels are not covered. To generalize,
let $x_j = 1$ indicate that column $j$ is selected and $x_j = 0$ that it is not.
Chapter 11 Set-cover Method

For a subset of subdomains to cover all labels the following must hold:

\[ \sum_{j=1}^{n} a_{1,j} x_j \geq 1 \]
\[ \sum_{j=1}^{n} a_{2,j} x_j \geq 1 \]
\[ \vdots \]
\[ \sum_{j=1}^{n} a_{m,j} x_j \geq 1 \]

Minimizing the number of selected columns, \( \sum_{j=1}^{n} x_j \), leads to a minimal partition. For example, if a new matrix is constructed from the columns of \( A_1, A_4, A_9, \) and \( A_{12} \) the following matrix \( A' \) is obtained.

\[
A' = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

For each row in \( A' \) the sum is 1. In fact, \( A' \) corresponds to a minimal partition, since there is no other combination of fewer columns of \( A \) that still covers all labels. To conclude, in order to find a minimal partition one need to select a number of columns from the partition matrix such that:

1. the sum of each row of is greater than or equal to 1,
2. the number of selected columns is minimized.

The selected columns must be a minimal partition according to Meudec’s definition, since criteria 1 guarantees that all labels are represented and criteria 2 forces the partition to contain a minimal number of subdomains.

Thus, in a more compact notation the problem can be stated as follows. Given a partition \( P \) with a set of labels \( L = \text{labels}(P) \) a minimal partition
11.2 Column Generation

$M$ is defined by:

$$
\begin{align*}
\text{minimize} & \quad z = 1^T x \\
\text{subject to} & \quad Ax \geq 1 \\
& \quad l \leq x \leq u \\
& \quad x_i \in \{0, 1\},
\end{align*}
$$

where $A$ is an $m \times n$-matrix, with $A_{i,j} \in \{0, 1\}$, $m = |L|$, and $n = |P|$, and where $x$ is a vector of length $n$. The upper and lower bound on the number of subdomains ($u$ and $l$) can be omitted if they are not known.

11.2 Column Generation

A great disadvantage with the set-cover method is that the initial partition must first be fully expanded in order to construct the Simplex matrix $A$. To expand $P$ may take a considerable amount of time and space. The space complexity of storing $A$, for example, is $O(|L||P|)$. Consequently, the Simplex matrix $A$ and the vector $x$ will be extremely large because $|P|$ is assumed to be very large. For example, if $E = a_1 \vee \cdots \vee a_k$, $k > 0$, then $|P(E)| = 2^k - 1$.

Consider the standard form of LP,

$$
\begin{align*}
\text{minimize} & \quad z = (c_1, \ldots, c_n)^T x \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0,
\end{align*}
$$

where $A$ is an $m \times n$ matrix with $m \leq n$. The foundations of LP are based on two important facts [42]. First, if the problem has a finite optimal solution, then it is always a corner point (extreme point) of the polyhedron defined by the linear constraints in $Ax = b$ and $x \geq 0$. Second, each corner point corresponds to an invertible $m \times m$ basis matrix $B$, which consists of any $m$ linear independent columns of $A$. Due to degeneracy two bases $B_i$ and $B_j$ may correspond to the same corner point.

Based on these two facts, the simplex method enumerates adjacent bases until an optimal value is found. Simply put, in each iteration the simplex method determines if the current basis matrix $B$ represents an optimal solution. If it is optimal the current solution is returned. However, if it is not, a new basis is formed by exchanging a column of $B$ with one of the columns of $N$, where $N$ contains the columns of $A$ that are not in $B$.

The optimality test is performed by considering the so-called reduced costs $\hat{c}_j = c_j - y^T A_j$ for all $j$ (or actually only for those $A_j$’s in $N$). If

105
\[ c_j \geq 0, \] then the current basis is optimal. Otherwise a variable \( x_t \) that satisfies \( \hat{c}_t < 0 \) is selected as entering variable. The entering variable represents a column \( A_t \) (which is currently found in \( N \)) that is to enter \( B \). The leaving variable (or column) is then determined using \( A_t \).

As can be seen from the rough outline above the simplex method does not require \( A \) to be explicitly available. All that is needed is a way to determine if \( B \) is optimal and to produce a column \( A_t \) that violates the optimality conditions. In some situations, when the coefficient matrix \( A \) has a specific structure, it is possible to find such an \( A_t \) without an explicit reference to \( A \).

In fact, this technique, known as column generation [42], is applicable in the set-cover method by generating the columns of \( A \) from a dependence-free, singleton partition graph. When column generation is integrated with branch-and-bound it is called branch-and-price [61]. It can be used to solve huge integer programs.

Let \( P \) be the partition to minimize. Let \( G \) correspond to the partition graph of \( \text{expand}_P(P) \). Since \( G \) is dependence-free all labels of \( P \) are also in \( G \). Additionally, \( G \) is also singleton meaning that all nodes contain exactly one label. Thus, the rows of \( A \) correspond to the union of the node labels of \( G \), and a column of \( A \) corresponds to a path through \( G \).

### 11.2.1 Preliminaries

Assume the standard form of LP in equation (5.1). The columns of \( A \) can be ordered such that \( A = (B, N) \), where \( B \) is the basis matrix and \( N \) is the non-basis matrix. In the same way \( x \) needs to be reordered such that

\[
x = \begin{pmatrix} x_B \\ x_N \end{pmatrix}
\]

where \( x_B \) is the vector of basic variables and \( x_N \) is the vector of nonbasic variables. The objective function can be written as

\[
z = c_B^T x_B + c_N^T x_N
\]

where \( c_B \) corresponds to the coefficients of the basic variables and \( c_N \) to the coefficients of the nonbasic variables.

The relaxed version of the set-cover problem (11.1), without bounds, can be stated as

\[
\begin{align*}
\text{minimize} & \quad z = 1^T x \\
\text{subject to} & \quad Ax \geq 1 \\
& \quad x \geq 0.
\end{align*}
\]
11.2 Column Generation

Using slack variables we can transform the problem above into the standard form of LP.

\[
\begin{align*}
\text{minimize} & \quad z = 1^T x + 0^T s \\
\text{subject to} & \quad (A, S) \begin{pmatrix} x \\ s \end{pmatrix} = 1 \\
& \quad x = (x_1, \ldots, x_n) \geq 0 \\
& \quad s = (x_{n+1}, \ldots, x_{n+m}) \geq 0.
\end{align*}
\]

and \( S = -I \), where \( I \) an identity matrix of size \( m \times m \) and \( s \) is a vector of slack variables of length \( m \). As previously stated \( m \) is the number of unique node labels of \( G \).

Given this, the simplex algorithm can be outlined as follows. The method starts with a basis matrix \( B \) corresponding to a basic feasible solution \( x_B = \hat{b} = B^{-1}b \geq 0 \).

1. First optimality is determined. Compute the vector \( y^T = c_B^T B^{-1} \).

   Compute the reduced costs \( \hat{c}_N = c_N - y^T N \). If \( \hat{c} \geq 0 \), then the current basis is optimal (return \( x \)). Otherwise, a variable \( x_t \) satisfying \( \hat{c}_t < 0 \) is selected as entering variable.

2. Given the entering variable \( x_t \) the leaving variable \( x_s \) can be determined using the column corresponding to \( x_t \).

3. Update basis matrix \( B \). The columns are now swapped.

Not having an explicit reference to \( A \) (specifically \( N \)) we must figure out a way to perform the optimality test. Steps 2 and 3 are left unaffected.

To determine whether the solution is optimal we compute the vector of simplex multipliers

\[ y^T = c_B^T B^{-1} \]

and verify whether the reduced cost \( \hat{c} \) is greater than or equal to 0. If \( y_i \) is negative, we know that the reduced cost of the \( i \):th slack variable \( x_{n+i} \) is

\[ \hat{c}_{n+i} = 0 - y^T (-e_i) = y_i < 0. \]

Here \( e_i \) is a vector of length \( m \) with a 1 in position \( i \) and zeroes elsewhere. Therefore, we can choose the slack variable as the entering variable. No further computation of the reduced costs coefficients are needed.

However, if \( y \geq 0 \) then the slack variables all have nonnegative reduced costs. Therefore, we must determine whether the original variables satisfy optimality. That is if

\[ \hat{c}_j = 1 - y^T A_j \geq 0, \quad j = 1, \ldots, n. \]
Chapter 11 Set-cover Method

To do this we form an auxiliary problem that finds the column $A_t$ with most negative reduced cost $\hat{c}_t$. If $\hat{c}_t < 0$ then $x_t$ is the entering variable, otherwise the current solution is optimal.

11.2.2 Auxiliary Problem – Longest Path

The most negative reduced cost $\hat{c}_t = 1 - y^T A_t$ is given when $y^T A_t$ is maximized. That is,

$$\text{maximize} \quad y_1 a_1 + y_2 a_2 + \cdots + y_m a_m.$$ 

Intuitively, each $A_j = (a_1, \cdots, a_n)$ represents a path through the singleton, dependence-free partition graph $G$ of $P$. That is, each $a_i$ corresponds to the usage (0/1) of label $i$ and each $y_i$ corresponds to a profit gained using that label. Certain $A_j$:s will have higher profit, meaning that the corresponding subdomains are more attractive to use as part of a minimal partition. The path with the highest profit is therefore the column with the most negative reduced cost.

Let $G = (V \cup \{s, t\}, E, D)$ where each edge $(v, w) \in E$ has been labeled with a distance equal to the profit of the node $w$. That is,

$$D = \{(e, y_w) \mid e = (v, w) \in E\}$$

The label of $s$ and $t$ is the empty label $\epsilon$ which has zero profit ($y_s = y_t = 0$).

By finding the longest path between $s$ and $t$ in $G$ we identify the $A_j$ with the most negative reduced cost. This problem is best solved by negating the edge profits and find the shortest path through the network. For acyclic graphs this very simple. Using Bellman’s Equations the shortest path can be computed in linear time, $O(|E|)$ (see reference [62] section 4.4).

The inequality below shows the upper and lower bound of the number of edges in a partition graph (including start and stop nodes).

$$|V| - 1 \leq |E| \leq \left\lfloor \frac{|V|^2}{4} \right\rfloor - 1$$

The lower bound is due to the fact that all nodes are connected by at least one incoming or outgoing edge. The upper bound is because the graph is arranged in levels with edges going only to higher levels. Fewer levels (normally) results in more edges. The number of edges is much less than the theoretical upper bound, because the nodes are typically
distributed over more than 4 levels (including the levels corresponding start and stop nodes). For example, the graph in figure 9.5 on page 82 has 24 nodes including start and stop nodes. The number of edges is 58. We have $23 \leq 58 \leq 143$.

The cost of relabeling of edges is equal to the number of edges $|E|$, however, with a simple modification edge costs can be stored in the nodes instead, because all outgoing edges from a node have the same cost. Thus, the running time of the auxiliary problem is $O(|E|+|V|)$.

11.3 Performance

Theoretical analysis of the performance of the set-cover method is difficult. A proper theoretical model of an average domain is hard to derive. Even if one existed, further analysis would be extremely challenging.

We suggest using software such as the CPLEX [63] Mixed Integer Optimizer to implement a solver for the set-cover problem. The mixed integer optimizer (presumably based on CPLEX’s Simplex optimizer) employs a branch & bound technique that takes advantage of cutting-edge strategies. Column generation, or some other decomposition technique, can be integrated CPLEX using its API.

Let $P$ be a partition, $L = \text{labels}(P)$ its set of labels and $N = \text{nodes}(P)$ its set of nodes. Let $M$ be a minimal partition of $P$. Let $G = (V, E)$ be the partition graph of $\text{expand}_P(P)$.

The time and space complexity given below are not absolute facts, but rather reasonable estimates, since the exact workings of CPLEX are unknown.

11.3.1 Time Complexity

IP problems (NP-hard) like this can be solved by relaxing the integer constraints on $x$. This results in a linear programming (LP) problem for which there are many algorithms, perhaps the most famous being the Simplex method (see chapter 5). If the relaxed problem does not produce a feasible integer solution, additional constraints can systematically be added in a recursive manner to constrain the violating variable to either 0 or 1. This technique is called branch & bound. [43]

LP problems can be solved in polynomial time, but the most widespread algorithm, the Simplex method, is theoretically exponential. In practice, however, the Simplex method often shows a linear performance (see chapter 5).
Chapter 11 Set-cover Method

The iteration complexity of the Simplex method using column generation is between $O(|L|^2 + |E| + |V|)$ and $O(|L|^3 + |L|^2 + |E| + |V|)$ depending on whether periodic refactorization is performed.

To get a perspective on the sizes of $L$, $E$, and $V$ we can do the following reasoning. Finding a minimal partition is of specific interest if $P$ is large. In situations when $P$ is small finding a minimal partition may mean unnecessary work.

Specifically, when $P$ is large and there is a big difference in size between $M$ and $P$ it is desirable to find $M$. In other situations, when $M$ and $P$ are of similar size one cannot afford to find a minimal partition. In other words, no minimization algorithm would perform well since there are too many dependent paths in $P$. Conclusively, if $M \sim P$, one has to settle for a reduced partition $R \subset P$.

Hence, a reasonable assumption is that $P$ is large and that $|M| \ll |P|$. In these situations $P$ does not contain many dependent paths, and therefore, the partial expansion of $P$ should not be much larger in size compared to no expansion.

Under the assumption that $|M| \ll |P|$ then the number of nodes $|V|$ is in the neighborhood of the number of labels $|L|$ (but larger). That is, $|L| \leq |V| \ll |P|$.

11.3.2 Space Complexity

Using column generation it is sufficient to construct only the basis matrix $B$ since $A$ is derived indirectly from $G$. This reduces the space complexity of representing the simplex matrix to $O(|L|^2)$. In the same manner vector $x$ can be represented only using $x_B$ which is of length $|L|$.

Space for storing the partially-expanded graph is also required. The assumption here is the same as for the time complexity, $|M| \ll |P|$, which means that the partially expanded graph is not much larger than the unexpanded one.
Chapter 12
Path-cover Method

— Many are stubborn in pursuit of the path they have chosen, few in pursuit of the goal.
Friedrich Nietzsche

The partition graph has a very interesting structure. The graph is directed and acyclic, where edges go from left to right. The nodes can be arranged in levels, where edges only connect adjacent levels. An acyclic network can always be arranged in this way by adding extra nodes.

In figure 12.1 we see the result of a partial expansion of expression (9.1) arranged in canonical form. The fully-expanded partition corresponds to all paths between the two nodes start (s) and stop (t). Nodes marked with the empty label (ε) are only used for level arrangement.

In this chapter we will learn that a minimal partition can be identified by formulating a network problem over a dependence-free, singleton partition graph. Specifically, the minimal partition problem can be modeled as minimum-cost flow problem with additional constraints to handle duplicate nodes. Duplicate node are nodes in the partition graph having the same label (see figure 12.1).

Duplicates can arise in two ways. First, they are introduced by the labeling function (see definition 8.4. A good labeling function should introduce duplicates since this means that the function has realized that two subexpression are equal. Second, nodes may be duplicated by expandP when trying to find a dependence-free, singleton partition graph.

Definition 12.1. A duplicate label of a partition P is a label represented in more than one subdomain of P. On the contrary, a unique label is a label which represented in exactly one domain of P.

Similarly, a duplicate node (unique node) in a partition graph G of a partition P is a node representing a duplicate label (unique label).
Figure 12.1: A partition graph in canonical form with start and stop nodes.
Theorem 12.1. Let $G = (N, A)$ be a partition graph. Then the number of paths in a minimal path-cover $M$ of $G$ is bounded by
\[
\max_{i \in L} |N_i - N^c| \leq |M| \leq \max_{i \in L} |N_i|,
\]
where $N_i$ denotes the set of nodes in level $i$, $L$ denotes the set of level indices in $G$ and $N^c$ denotes the set of duplicate nodes in $G$.

Proof. Let $N_i \in L$ be an arbitrary level of $G$. Since $G$ is a partition graph, it is directed and acyclic and by definition there are no arcs between nodes in the same level. Therefore, if $N_i$ does not contain duplicate nodes then exactly $|N_i|$ paths are required to cover the node names of $N_i$. If, however, $N_i$ has duplicate nodes at least $|N_i - N^c|$ paths (and at most $|N_i|$) are needed. Thus, the number of paths in $M$ must be bounded by
\[
\max_{i \in L} |N_i - N^c| \leq |M| \leq \max_{i \in L} |N_i|.
\]
Note that if $N_i \cap N^c = \emptyset$ then $N_i - N^c = N_i$. Thus, if at least one of the levels with highest node count only contains unique nodes then $|M|$ is bounded by equality.

Definition 12.2. Given a dependence-free, singleton partition graph (not necessarily containing padding nodes) with node set $N \cup \{s, t\}$ and arc set $A$, a minimal-path cover of a partition graph is a minimal set of paths from $s$ to $t$ such that

(i) each unique node must be covered by at least one path and

(ii) at least one of the nodes representing a duplicate label must be covered by at least one path.

In this chapter we show that the minimal-path cover problem can be modeled as a Minimum Cost Flow Problem (MCFP) with additional (complicating) constraints that express dependency between the flow through the arcs of duplicate nodes.

The arc flows are considered as the variables of the problem, and the flow on an arc is defined as the number of times this arc is used in some path from $s$ to $t$.

Omitting condition (ii) the problem is a pure MCFP, which can be solved as a linear problem by e.g. the Network Simplex Algorithm [62]. Any (extreme) solution to this linear problem has the property of being integer.

However, taking condition (ii) into consideration, the integrality property is destroyed. Thus, we cannot interpret a solution in terms of the
Chapter 12 Path-cover Method

Figure 12.2: Nodes are transformed to arcs to ensure coverage.

number of times each arc is used. Therefore we have to add integral-
ity constraints on the flow variables, thereby classifying the optimization
problem as an integer problem (NP-hard). These problems can be solved
by a technique known as branch & bound [43], that recursively creates
and solves a set of linear problems until an integer solution is found.

12.1 Construction of the Network

Start with \((N \cup \{s\} \cup \{t\}, A)\). Denote each group (pair, triplet, \ldots) of
nodes representing a duplicate label by \(N^c_l, l \in L\), where \(L\) is the set of
all labels. Let \(N^c := \cup_{l \in L} N^c_l\).

Since MCFP considers the arcs as variables and not the nodes, it is
necessary to split each node in \(N\) in two and add an arc between them.
In this way the flow over this arc is equivalent to the flow through the
origin node.

In figure 12.2 node \(n_i\) is split into the nodes \(n_i^o\) (origin) and \(n_i^d\) (desti-
tination). The arc \((i^o, i^d)\) is called internal arc. All arcs to node \(n_i\) are
mapped as incoming arcs to \(n_i^o\) and all arcs from node \(n_i\) are mapped as
outgoing from \(n_i^d\). Denote the new set of nodes by \(N^o\) and \(N^d\), respecti-
vively, and the new set of internal arcs by \(A^{od}\).

For each unique node \(i \in N - N^c\), the lower bound of the flow on arc
\((i^o, i^d)\) is set equal to one, that is, \(lb_{i^o,i^d} = 1\). Thus, each unique node will
be covered by at least one path. Let the lower bound for all other arc
flows be equal to zero.

Given theorem 12.1 we know the upper bound \(u\) and the lower bound
\(l\) on the number of paths needed in the final solution. Since only the
bounds are known and not an exact number, the supply at node \(s\) (as
well as the demand at node \(t\)) is variable. This can be modeled by adding
a single arc \((t, s)\) to the network, with a cost equal to 1. All other arc
costs are equal to zero. Set the upper and lower bound on \((t, s)\) to \(u\) and
\(l\), that is, let \(lb_{ts} = l\) and \(ub_{ts} = u\).

The constructed network consists of the node set \(N' = N^o \cup N^d \cup \{s\} \cup
\{t\}\) and the arc set \(A' = A \cup A^{od} \cup \{(t, s)\}\).

A minimum cost (circular) flow in the constructed network has the
12.2 Fixation of Duplicate Nodes

property of being integer. Thus, it correctly represents a minimal set of paths from \( s \) to \( t \) in the original network. However, it is not guaranteed that the nodes in \( N^c \) are covered.

For each \( l \in L \), in order to force at least one of the nodes in \( N^c_l \) to be covered by at least one path we have to add such complicating constraints to \( \text{mcfp} \). Consider the following mathematical formulation of the problem.

\[
\text{mcfp}^{+} \quad \text{(ii)}
\]

\[
\begin{align*}
\min & \quad c^T x \\
\text{s.t.} & \quad A^1 x = b \quad \text{node balancing constraints} \\
& \quad A^2 x \geq 1 \quad \text{constraints for duplicate nodes} \\
& \quad x \geq 1 \quad \text{lower bounds} = 1 \text{ for unique nodes}
\end{align*}
\]

where \( A^1 \) is the node incident matrix for the network \((N', A')\) and \( A^2 \) the coefficient matrix for the complicating constraints. The vector \( c \) contains a one in the position of arc \((t, s)\), all other entries are equal to zero. A complicating constraint for a specific label is formulated as (for two copies):

\[
x_{i_0j^d} + x_{j_0j^d} \geq 1
\]

meaning that the sum of the flow through a group of duplicate nodes must be at least one.

12.2 Fixation of Duplicate Nodes

An upper and lower bound of a minimal cover for a partition graph can be calculated recursively from its partition tree (by induction of theorem 12.1). In the illustration below we denote the upper and lower bound of a partition tree \( t \) by \(^1t^u\). For a unique node \( n_u \) and a duplicate node \( n_d \) the bounds are \(^1n^1_u\) and \(^0n^1_d\).

\[
\begin{array}{c}
\text{l}_1 + \text{l}_2 + \text{u}_1 + \text{u}_2 \\
\text{l}_1 \text{t}_1 \text{u}_1 \quad \text{l}_2 \text{t}_2 \text{u}_2
\end{array}
\]

\[
\begin{array}{c}
\text{max}\{\text{l}_1, \text{l}_2\} \cdot \text{max}\{\text{u}_1, \text{u}_2\} \\
\text{l}_1 \text{t}_1 \text{u}_1 \quad \text{l}_2 \text{t}_2 \text{u}_2
\end{array}
\]

There are situations when it can be proved (using theorem 12.1) that a certain duplicate node can be treated as if it is unique, without affecting the outcome of a minimal cover. Such a duplicate is referred to as pseudo-unique. Specifically, given the multiplication tree above, if \( l_1 \geq u_2 \) (or \( l_2 \geq u_1 \)) then all duplicate nodes in \( t_2 \) (or \( t_1 \)) are pseudo-unique.
A node being pseudo-unique means that its lower bound (on the internal arc) can be set to 1, implying that the complicating constraint for its duplicate group can be removed. In this way we can greatly reduce the complexity of the network problem. If all complicating constraints can be removed the problem is pure MCFP.

For example, consider the partition graph in figure 12.1 and the subgraph \( S_1 = S_2 \cdot S_3 \) where \( S_2 = e_1 + e_2 \) and \( S_3 = d_2 d_3 + d_2 d_4 + d_1 d_3 + d_1 d_4 \). Using theorem 12.1 it can be verified that the lower and upper bounds \( S_i(l, u) \) of the \( S_i \)'s are \( S_1(4, 4) \), \( S_2(0, 2) \), and \( S_3(4, 4) \). Since the lower bound for \( S_3 \) is greater than or equal to the upper bound for \( S_2 \) we can force a path through each duplicate in \( S_2 \) without affecting the bounds on \( S_1 \).

The function bounds in listing 12.1 calculates the lower and upper bounds on the number of paths in a minimal partition represented by a given partition tree \( t \). Specifically, bounds returns a tuple \( (l, u, R, F) \), where \( l \) and \( u \) is the lower and upper bound. The so-called required nodes \( R \) is a set of unique and pseudo-unique nodes all having their lower bound (of the internal arc) set to 1. The set of node \( F \), called the free nodes, consists of the duplicate nodes that are not required, that is, that are not pseudo-unique. Note that \( N = R \cup D \).

### 12.3 Solving the Network

We suggest using the CPLEX [63] optimization system to solve the network problem (MCFP+(ii)). Not only is CPLEX based on linear programming (LP) and branch & bound (BB), but it also utilizes the network structure of an LP problem and can handle huge problem sizes.

If it can be shown that CPLEX cannot handle the generated problem instances, one should develop a tailored solution based on a decomposition technique, e.g. column generation [42]. Such a solution can be integrated with CPLEX using its API.

Alternatively, we suggest to modify the traditional BB strategy where branching is performed on the problem variables, in this case the node flow, and instead branching on the lower bounds of the node flows. This ensures that the height of the BB solution tree is less than the number of duplicate node groups and that each subproblem in the tree has an integer solution. The algorithm given below can easily be modified to call either CPLEX or a tailored network solver.

Assume that duplicate groups \( V^c_i \) are numbered from 1 to \( m \).

1. Let \( P_0 \) be the network problem without complicating constraints.
Listing 12.1: An algorithm for determining arc bounds.

```
function bounds(t : Tree) : (l, u, R, D)
    if isAtom(t) then
        if isUnique(t) then
            return (1, 1, {t}, ∅)
        else
            return (0, 1, ∅, {t})
        end
    else if isPlus(t) then
        (l1, u1, R1, D1) := bounds(t.left)
        (l2, u2, R2, D2) := bounds(t.right)
        return (l1 + l2, u1 + u2, R1 ∪ R2, D1 ∪ D2)
    else
        (l1, u1, R1, D1) := bounds(t.left)
        (l2, u2, R2, D2) := bounds(t.right)
        l := max {l1, l2}
        u := max {u1, u2}
        R := R1 ∪ R2
        if l1 ≥ u2 then
            return (l, u, R ∪ D2, D1)
        else if l2 ≥ u1 then
            return (l, u, R ∪ D1, D2)
        else
            return (l, u, R, D1 ∪ D2)
        end
    end
end bounds
```

2. Solve $P_j$ using some network algorithm.

3. If the solution does not satisfy the complicating constraints for some group of duplicates $N^c_i$ then for each node $n \in N^c_i$ solve a subproblem $P_n$ by recursing to step 2. $P_n$ is given by $P_j$ but the lower bound of the flow over $n$ set to 1. Return the best of the sub-solutions. Note that the current best solution can be used to cut off branches in the solution tree.

4. If the solution satisfies the complicating constraints then return the current solution.

In each recursion one duplicate is fixated (made pseudo-unique) and thus it is guaranteed that its complicating constraint is satisfied. Two
or more nodes in a group $N_i^c$ never need to be fixated simultaneously, because this is done indirectly through the fixation of some other node in a group $N_j^c$. It can be shown that the maximum number of network problems $P_j$ is equal to

$$1 + \sum_{i=2}^{m} \prod_{j=1}^{i-1} |N_j^c|$$

where $m$ is the number of duplicate groups.

Each new subproblem can effectively use the previous solution $P_j$ to create a new basic feasible solution by adding a path that connects the fixated duplicate with $s$ and $t$. Any path will do, but the best choice is probably the longest path or the path that traverses the most uncovered duplicates since such a path ought to increase the chance of cut-offs in the BB tree.

### 12.4 Performance

As with the set-cover method, a theoretical analysis of the performance of the path-cover method is difficult. A proper theoretical model of an average domain is difficult to derive. Even if one existed further analysis would be extremely challenging.

The network simplex method [42, 62, 64] is a version of the simplex method tailored for network problems. It requires fewer operations per iteration than the standard simplex method. However, the upper bound of number of iterations is still exponential, though practical results show that the simplex needs between $m$ and $3m$ iterations to find a solution for problems on standard form.

Therefore, the performance of the path-cover method is expected to be better than that of the set-cover method.
Chapter 13
Dag-based Partial Expansion

Node duplication is the price paid for the simplicity of the tree-based partial expansion rules. Whenever two dependent subtrees are combined, copies of these trees are created. This means that the resulting partially expanded partition will contain duplicate nodes.

Duplicate nodes increase the level of difficulty in finding a minimal path cover. First of all, considering the path-cover approach for finding minimal a partition in chapter 12, the extreme points of a unimodular matrix are all integer, meaning that a minimal path-cover can be solved in polynomial time. The complicating constraint on the other hand can force the constraint matrix to be non-unimodular and thus a branch-and-bound technique may have to be applied.

Secondly, the partition graph becomes larger than necessary, which in turn affects the overall performance of the algorithms based on partially expanded partition. Therefore, if the duplicates can be eliminated much would be gained.

A solution to the duplicate node problem would be to remove duplicates after partial expansion has finished. It can be seen that by a few transformations of the graph given in figure 12.1, duplicate nodes introduced by partial expansion can be eliminated resulting in the graph in figure 9.5.

A better solution, though, is to avoid the creation of duplicate nodes in the first place. This is the approach taken in this chapter by enhancing the partial expansion by so-called dag rules.
Chapter 13 Dag-based Partial Expansion

13.1 Overview

The idea behind the dag rules is to postpone expansion (node duplication) of the partition graph as long as possible. For example, consider the expansion of $a_1(t_1 + a_2 + t_2)$ shown in figure 13.1, where $a_i$ are dependent atoms and $t_i$ represent independent subtrees. Using the tree rules the label $a_1$ is duplicated in step 1. The dag rules, on the other hand, avoid duplicate labels by reusing partition tree parts that do not yet have to be expanded. Therefore, the tree is no longer a tree but a directed acyclic graph (dag). For example, using the dag rules the initial tree is transformed according to figure 13.2.

The second dag contains two operator nodes marked $A$ and $B$. These are referred to as direct and indirect parents of $a_1$. A parent being direct or indirect is only a matter of how the dag is viewed. In the case above it is more natural to consider $A$ as the direct parent and $B$ as the indirect. A node can only have one direct parent. All other parents are indirect. Thus, once we have settled that, say, $A$ is the direct parent of $a_1$, then all other parents of $a_1$ (including $B$) are indirect.

13.2 The Dag Rules

As with the tree rules the predicate $\text{dep}(t_1, t_2)$ is true iff the two dags are dependent. The relation is not transitive. The left and right subdag of the top node is referred to as $l$ and $r$ respectively. Addition nodes are denoted by $+$, multiplication nodes are denoted by $\cdot$, and atoms (labels) are denoted by $a$. Arbitrary dags are denoted by $t_i$. A circled node in the right hand side of a rewrite rule indicates a recursive call to expand $P$. In case of several recursive calls, the invocation order is left to right and bottom to top.

Since the dag rules operate on dags a node can have more than one parent. Therefore it is important that the identity of the nodes are preserved as long as possible. In this way unwanted node duplication can be avoided. Therefore, some of the nodes in a rule, which do not already have a unique name, are marked with a variable subscript in order to get a unique handle referencing that node.

For example, as can be seen in the rules below, the root node of an incoming dag must be the same as the root node of an outgoing dag (actually they only need to have the same parents).

The dag rules are still constructed using the same five basic rewrites described on page 93. However, care must be taken, because the com-
13.2 The Dag Rules

\[
\begin{align*}
  a_1 + &t_1 + a_2 t_2 \\
  \rightarrow &+ \\
  (R3) &+ \\
  a_1 &t_1 + a_2 t_2 \\
  \rightarrow &+ \\
  (R3) \\
  a_1 &t_1 + a_2 &t_2
\end{align*}
\]

Figure 13.1: Applying the tree rules on \(a_1(t_1 + a_2 + t_2)\).

\[
\begin{align*}
  a_1 + &t_1 + a_1 t_2 + a_12 + t_2 \\
  \rightarrow &+ \\
  (R2.3) &+ \\
  a_1 &t_1 + a_2 &t_2 \\
  \rightarrow &+ \\
  (R2.3) \\
  a_1 &t_1 + a_12 &t_2
\end{align*}
\]

Figure 13.2: Applying the dag rules on \(a_1(t_1 + a_2 + t_2)\).
mutative rule is not necessarily domain-preserving when applied to dags. Therefore, one objective in transforming the tree rules is to eliminate commutation to avoid complications.

The other basic rewrites of identity, creative, distributive, and associative, are domain-preserving for dags. Specifically, the last two are also graph preserving.

**Definition 13.1.** Let $T$ be a partition dag and denote its partition graph by $G_T$. Then a rewrite rule $r$ is called *graph preserving* if an application of $r$ on a subdag of $T$ results in a new dag $T'$ such that $G_{T'} = G_T$.

By definition a graph-preserving rule is domain-preserving. This property is particularly desirable because even though the rewrite dramatically changes the shape of the partition dag, it does not affect the partition graph at all.

\[
\begin{align*}
    a & \longrightarrow a \\
    a_1 & \mathrel{R} a_2 \\
    a_1 & \mathrel{dep(l,r)} \longrightarrow \text{merge}(a_1, a_2) \mathrel{R} a_1 a_2
\end{align*}
\]
13.2 The Dag Rules

(R2.5) \[
\begin{array}{c}
\vdots
\end{array}
\]

(R2.6) \[
\begin{array}{c}
\vdots
\end{array}
\]

(R2.7) \[
\begin{array}{c}
\vdots
\end{array}
\]

(R2.8) \[
\begin{array}{c}
\vdots
\end{array}
\]
Chapter 13 Dag-based Partial Expansion

\[ R_{a/+} \cdot A_{t_1 \rightarrow t_2} \]

\[ \neg \text{dep}(l,t_2) \quad \text{dep}(l,t_1) \]

\[ \rightarrow \]

\[ R_{a/+} \cdot A_{t_2 \rightarrow t_1} \rightarrow t_1 \leftarrow t_2 \]

\[ \neg \text{dep}(l,t_1) \quad \text{dep}(l,t_2) \]

\[ \rightarrow \]

\[ R_{+/-} \cdot R_{t_2} \rightarrow t_1 \leftarrow t_2 \]

\[ \text{(R2.9)} \]

\[ \text{(R2.10)} \]

13.3 Rationale

The dag rules are very interesting. The majority of the rules are graph-preserving (8 of 10), meaning that in most rules the graph is not modified. Only rule (R2.2) and (R2.9) actually change the layout of the graph.

The graph-preserving rules recursively divide the graph into subgraphs (not disjunct). In other words, these rules almost work as a variable-sized peep hole, systematically scanning the partition graph for the two patterns of rules (R2.2) and (R2.9).

(R2.1) – Identity This rule is the same as the tree rule. The rule is graph-preserving.

(R2.2) – Creative In this rule the root node is replaced by an atom formed from the merge of \( a_1 \) and \( a_2 \). Additionally, it shows that the old nodes \( a_1 \) and \( a_2 \) are left as is, in case they have indirect parents created by, for example, rule (R2.3). The rule is not graph-preserving because of the newly created node. However, it is domain-preserving.

(R2.3) – Distributive The copy of \( a \) created in the right subdag of the root is avoided by a direct reference to the same node in the left subdag. This is the reason why dags are created by the dag rules and not trees.

Note that the old subdag \( A \) is left as is, meaning that it has become an indirect parent of \( t_1 \) and \( t_2 \). This relationship is indicated by the dashed arrows.

The rule is graph-preserving. This means that the partition graph of the left hand side (LHS) and the right hand side (RHS) are identical. The
partition graph can be derived from the dag in three simple steps. First, each multiplication expression $t_1 \cdot t_2$ that is the right child of parent $r$ and the left child of a parent $l$ is transformed as follows:

Second, each addition expression $A = t_1 + t_2$ is transformed accordingly:

Finally, remove all $\epsilon$ nodes of the graph from the graph by connecting each incoming neighbor of an $\epsilon$ node with each outgoing neighbor of the $\epsilon$ node.

Thus the LHS of rule (R2.3) is derived as follows. Node $p$ represents an indirect parent of dag node $A$. Indirect parents of other nodes are not of interest.

Repeating the same procedure but this time for the RHS results in:

**(R2.4) – Distributive** The tree version of this rule made a commutative rewrite of the left and right subtrees. Unfortunately, commutations are not always domain-preserving when working with dags. The intention of the tree rule was to swap left and right subtree and expand the root node again, in order for rule (R5) to by applied in the next step. However, a graph-preserving rule with the desired effect can be achieved by doing the distribution directly in this step instead.
This rule, as well as the next, applies to partitions of the form \((t_1 + t_2)(t_3 + t_4)\), where the two subdags \(t_1 + t_2\) and \(t_3 + t_4\) are dependent. Table 13.1 enumerates all such situations. The columns labeled \(t_i\) holds the dependence class of \(t_i\) in particular situation, where \(a\) is a dependence class and \(*\) is wildcard meaning any other dependence class then \(a\).

For example, situation 3 is where \(t_1, t_2,\) and \(t_4\) belongs to the same dependence class and \(t_3\) to another. By saying that a subdag belongs to a dependence class \(a\) we mean that the subdag contains an atom belonging to \(a\).

Basically there are two rewrites to consider: either the one of rule (R2.4) or rule (R2.5), that is,

\[
t_1(t_3 + t_4) + t_2(t_3 + t_4) \quad \text{or} \quad (t_1 + t_2)t_3 + (t_1 + t_2)t_4.
\]

The following two columns show the result of rewriting the expression according to each of two rules. In the last column the optimal choice of rule is given. The optimal choice is decided on how large portion of the expression that needs to be successively rewritten to reach a merge of the depending atoms. Less is better. For example, in situation 2, given a rewrite using rule (R2.4) both \(a_1(a_3 + *_4)\) and \(a_2(a_3 + *_4)\) needs to be expanded to clear all dependencies. However, if rule (R2.5) is used, only the dag \((a_1 + a_2)a_3\) needs to be expanded, since \((a_1 + a_2)\) and \(*_4\) are not dependent. Therefore an optimal choice is to use rule (R2.5). In other situations, like situation 7, both rules are as good as the other.

Consequently, by adding the condition \(\text{dep}(l, r) \land \neg \text{dep}(t_1, t_2)\) to (R2.4) it will handle situations 4–9. The rule is graph preserving.

(R2.5) – Distributive  Due to the sequential processing of the rules, it is sufficient that the dependence condition of this rule is \(\text{dep}(l, r)\). A more precise condition would be \(\text{dep}(l, r) \land (\text{dep}(t_1, t_2, t_3) \lor \text{dep}(t_1, t_2, t_4))\). This rule handles situations 1–3 of table 13.1. The rule is graph-preserving.

(R2.6) – Distributive  Previously this rule used a commutative rewrite, relying on (R3) for further expansion. Now the same effect is achieved with a distributive rewrite instead. The rule is graph-preserving.

(R2.7) – Right Associative  This rule is the dag version of the tree rule. The rule is graph-preserving.
13.4 The Final Piece in the Puzzle

Table 13.1: Optimal choice of distributive rewrite for \((t_1 + t_2) \cdot (t_3 + t_4)\) when left and right subtrees are dependent. The symbol \(a\) is a dependence class and \(*\) is a wildcard class other than \(a\).

<table>
<thead>
<tr>
<th></th>
<th>(t_1t_2t_3t_4)</th>
<th>(R2.4)</th>
<th>(R2.5)</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a a a a)</td>
<td>(a_1(a_3 + a_4) + a_2(a_3 + a_4))</td>
<td>((a_1 + a_2)a_3 + (a_1 + a_2))</td>
<td>any</td>
</tr>
<tr>
<td>2</td>
<td>(a a a *)</td>
<td>(a_1(a_3 + <em>_4) + a_2(</em>_3 + a_4))</td>
<td>((a_1 + a_2)) ((a_1 + a_2)*_4)</td>
<td>(R2.5)</td>
</tr>
<tr>
<td>3</td>
<td>(a a * a)</td>
<td>(a_1(<em>_3 + a_4) + a_2(</em>_3 + a_4))</td>
<td>((a_1 + a_2)*_3 + (a_1 + a_2))</td>
<td>(R2.5)</td>
</tr>
<tr>
<td>4</td>
<td>(a * a a)</td>
<td>(a_1(a_3 + a_4) + *_2(a_3 + a_4))</td>
<td>((a_1 + *_2)a_3 + (a_1 + *_2))</td>
<td>(R2.4)</td>
</tr>
<tr>
<td>5</td>
<td>(* a a a)</td>
<td>(*_1(a_3 + a_4) + a_2(a_3 + a_4))</td>
<td>((+_1 + a_2)a_3 + (+_1 + a_2))</td>
<td>(R2.4)</td>
</tr>
<tr>
<td>6</td>
<td>(a * a *)</td>
<td>(a_1(a_3 + *_4) + *_2(a_3 + *_4))</td>
<td>((a_1 + *_2)a_3 + (a_1 + <em>_2)</em>_4)</td>
<td>any</td>
</tr>
<tr>
<td>7</td>
<td>(a * * a)</td>
<td>(a_1(*_3 + a_4) + <em>_2(</em>_3 + a_4))</td>
<td>((a_1 + <em>_2)</em>_3 + (a_1 + *_2))</td>
<td>any</td>
</tr>
<tr>
<td>8</td>
<td>(* a * a)</td>
<td>(<em>_1(</em>_3 + a_4) + a_2(*_3 + a_4))</td>
<td>((+_1 + a_2)*_3 + (+_1 + a_2))</td>
<td>any</td>
</tr>
<tr>
<td>9</td>
<td>(* a a *)</td>
<td>(*_1(a_3 + *_4) + a_2(a_3 + *_4))</td>
<td>((+_1 + a_2)a_3 + (+_1 + a_2)*_4)</td>
<td>any</td>
</tr>
</tbody>
</table>

(R2.8) – Left Associative This rule is the dag version of the tree rule. The rule is graph-preserving.

(R2.9) – Commutative This rule is interesting. The nodes \(p_i\) denote parents of \(t_i\), if any, other than \(A\). The two subtrees are swapped in order for rule (R2.8) to match. Note that parents of old \(t_1\) \((t_2)\) are now parents of \(t_2\) \((t_1)\).

Since the parent node \(p_1\) has been formed using the other expansion rules it means that \(t_1\) must be the left child \(p_1\), and thus, the subgraph \(p_1\) is an incoming neighbor of \(t_1\). Similarly, \(t_2\) must be the right child of \(t_2\), and thus, the subgraph \(p_2\) is an outgoing neighbor of \(t_2\). Therefore, as can be seen from the partition graph below, this rewrite rule is domain-preserving (but not graph-preserving).

(R2.10) – Identity This rule is the dag version of the tree rule. The rule is graph-preserving.

13.4 The Final Piece in the Puzzle

To conclude, the dag-rules are the final piece to the puzzle in merging dependent nodes of a partition graph. Since substructures are reused, no new duplicate nodes are introduced in the graph. This means that
Chapter 13  Dag-based Partial Expansion

duplicates are only introduced by the labeling algorithm (just as it should be).

Expanding the partition in figure 9.4 using the tree-based rules results in the partition in figure 12.1.

If the dag rules are used the result is as given in figure 9.5. This is precisely the graph that Meudec failed to systematically derive.
Chapter 14

When Minimal Partitions Are Too Large

— Beware of bugs in the above code; I have only proved it correct, not tried it.
  Donald Knuth

There are situations in which a minimal partition would still be too large to be considered feasible to handle. For instance, when all variables used in a complicated specification belong to the same dependence class, a minimal partition is the same as a complete partition. For these particular specifications $\text{expand}_P(P) = \text{expand}_F(P)$. Therefore, we must have some alternatives to minimal partition testing.

In fact, it is quite easy to define adequacy criteria based on the partition graphs using control-flow based criteria.

1. *Path coverage* corresponds to all paths through the partition graph, that is, complete partition coverage.

2. *Reduced path coverage* corresponds to minimal partition coverage. Reduced path coverage is not an existing control-flow criteria.

3. *Branch coverage* corresponds to a reduced partition that traverses every branch in the partition graph.

4. *Statement coverage or node coverage* represents a reduced partition that covers all nodes of the partition graph.

Currently, it is not clear whether there are other control-flow based adequacy criteria, such as *multiple-decision coverage* or *condition coverage*, which can be transformed to a partition criteria.
Chapter 14 When Minimal Partitions Are Too Large

Study the partition graph shown in figure 14.1. Assume that all nodes belong to the same dependence class, that is, a minimal partition would require all 128 paths. However, considering branch coverage, it is easy to argue that it can be achieved by at least 16 paths. To be more specific, we must find a (minimal) set of paths such that all edges between level 3 and 4 are traversed at least once, since those are the two neighboring levels that have the most edges. In fact, the set of paths given in figure 14.1(c) is a minimal set of paths for branch coverage.

It is interesting to note that with a simple manipulation of the partition skeleton we can reduce the number of paths required for branch coverage. Consider moving the subdomain \{17, 18\} to the middle of the skeleton expression (figure 14.2(a)). This results in a graph with fewer edges. Here we see that a minimum set of paths for branch coverage must have at least 8 paths. Such a set is given in figure 14.2(c).
Figure 14.1: A partition with 128 paths. At least 16 paths needed for branch coverage.
Figure 14.2: Simply by changing the order of the subdomains this graph now only requires 8 paths for complete branch coverage.
Chapter 15
Conclusions and Future Work

— When I examine myself and my methods of thought, I come to the conclusion that the gift of fantasy has meant more to me than any talent for abstract, positive thinking.

Albert Einstein

Meudec’s automatic partitioning strategy is a highly complex process. The input domain of a specification is partitioned into a set of subdomains using a set of partitioning rules. The generated set of subdomains is referred to as the complete partition. Yet, because of combinatorial explosion of the number of subdomains in the complete partition, Meudec’s strategy would have little practical applicability, unless an efficient way to reduce its size is identified.

One way to reduce the size of the complete partition is to use a better set of partition rules, where better rules implies rules producing smaller partitions. Another way, chosen by Meudec in his dissertation, is to derive a reduced partition based on the subdomains in the complete partition. His approach is based on a full expansion of the partition graph, which is a compact representation of a partition, resulting in a combinatorially large set of subdomains, which is then reduced to a so-called minimal partition.

In this thesis Meudec’s ideas concerning partition reduction are further developed. It is shown that Meudec’s minimal-partition algorithm is incorrect and does not always produce a minimal partition.

Instead, it is shown that Meudec’s set-based definition of a minimal partition can be modeled as an optimization problem. Two such models are given, both based on integer programming. One is based on set-covers and the other is based on path-covers in network flows.

Specifying the minimal partition problem as an optimization problem is advantageous because there are well-known algorithms that solve such
problems, for example, the (network) simplex method. We suggest using CPLEX, which is a software system for solving optimization problems based on linear programming and so-called branch & bound. CPLEX has the ability to utilize the network structure of a problem and can handle huge problem sizes.

Constructing the two models required a new theory for removing dependence among the subdomains of a partition to be developed. In fact, it was already pointed out by Meudec in his thesis that finding a dependence-free partition could lead to a better algorithm. However, he failed to find a solution to the problem. This is the reason for why his algorithm was based on the so-called fully expanded partition, because this partition is always dependence-free.

In this work a technique for removing partition dependence is developed. In short, the algorithm partially expands dependent parts of a partition. For each expansion the partition becomes less and less dependent. In contrast to Meudec’s minimal algorithm, which needs a fully expanded partition, this technique is much more efficient, since it only expands as much as is necessary.

Thus, an arbitrary partition can be minimized in two steps: (1) find a dependence-free partition through partial expansion, (2) apply either set-cover or path-cover method to find a minimal partition.

The work with the partial expansion algorithm has also led to other interesting results besides solving the minimal partition problem. It is shown how to determined a lower and an upper bound on the number of subdomains in a minimal partition by analyzing dependence-free partitions. Consequently, this has led to a stronger definition of the two to minimal partition methods.

A drawback of Meudec’s partition method is that for ill-conditioned partitions, that is, partitions with maximum or very high dependence, the process of finding a dependence-free partition is equal to full or close-to-full expansion. Therefore, a there is also a discussion on alternative approaches when even minimal partitions are too large or the complete partition is ill-conditioned. These approaches are based on ideas from control-flow based adequacy criteria.

\section*{15.1 Future Work}

An actual implementation of automatic partitioning for some specification language is desirable. This would not only let us evaluate the performance of minimal-partition techniques described in this work, but also allow us
to collect empirical data on the composition of normal partitions, which in turn is important for devising high-performance reduction algorithms. An implementation may also be used to assess effectiveness of test cases generated from the partitions derived with automatic partition testing. For instance, the strength of test cases can be compared based on complete partition coverage, minimal partition coverage, partition branch coverage, partition node coverage etc.

A more in-depth study of non-minimal techniques, such as partition branch coverage, is appealing. For instance, we have seen that a simple rearrangement of the nodes in a partition graph can result in a partition that requires half the number of tests for branch coverage than the original partition. The results found here may have an impact on minimal techniques as well.

Finally, an investigation of Meudec’s partition rules would be valuable. It may be possible to generate tighter partitions, while still covering a minimal partition condensed from a complete partition generated with the original rules.
Part III

Testing-tool Design
Chapter 16

Testing-tool Design

— The test for whether or not you can hold a job should not be the arrangement of your chromosomes.

*Bella Abzug*

In the final part of this thesis two testing tool prototypes with different designs are studied. The tools have been designed with simplicity in mind, therefore they are assumed to be used with, but not limited to, random input generation.

The first prototype, shown in figure 16.1, is of a type of testing tool referred to as serial and specification-based. The latter property means that the tool uses a specification to verify test outcome.

Programs and specifications are assumed to have been written in the same language. This allows for many of the components of the tools to be reused, with little or no modification, when applied on either programs or specifications.

Any language can be used as a specification language; for instance, one could use a traditional programming language such as Ada, C, or Java. The difference between a programming language such as Java and a specification language, say *VDM-SL*, is that the latter has far more built-in semantics that can be utilized in partition testing. For example, consider the expression $\exists x \in a : r = x$, which asserts that the return value $r$ is a member of the given list. In Java this can be written as in listing 16.1.

Applying Meudec’s coverage criterion, for example, on the logical specification above results in 5 subdomains (2 of which are, however, infeasible), while statement coverage and branch coverage on the program code only result in 1 and 2 respectively.
Chapter 16 Testing-tool Design

Listing 16.1: A program that returns true iff \( r \) is a member of \( a \).

```java
boolean contains(Object[] a, Object r) {
    for (int i = 0; i < a.length; i++) {
        if (r == a[i]) {
            return true;
        }
    }
    return false;
}
```

Assuming that specifications are written in a programming language, the questions studied in this part are: (1) if a program-based coverage criteria provides enough granularity for the generated tests, and (2) how is this affected when using random input generation?

There is an alternative implementation of the specification-based tool shown in 16.2, referred to as a parallel, specification-based testing tool. However, this design is not suitable when using random testing for efficiency reasons. More on this topic is discussed in section 18.6.

For some problems it is hard to write a specification. In such situations one can use a reference program to verify the program result. In many situations there naturally exist two or more versions of a program. The reference program could be a prototype, an older program version (when performing regression testing [65,66]), a version of the program compiled for a different platform, or a version run in a different environment, for example, when testing for the Year 2000 Problem [67].

Therefore, the second tool, shown in figure 16.3, uses a reference program to verify test outcome. The tool is referred to as a program-based testing tool or reference-based testing tool.

Since specifications often express what to do, in contrast to programs expressing how to do it, it is of interest to find out how the fault-detection effectiveness is affected by using a program-based testing tool.

The rest of the work is organized in the following way. Section 16.1 presents related work. In chapter 17 a brief introduction to testing concepts is given. In chapter 18 the specification-based testing tool is described and analyzed. Similarly, chapter 19 is dedicated to the program-based testing tool. Finally, chapter 20 presents conclusions from part III and proposes future work.
Figure 16.1: Serial, specification-based testing tool.

Figure 16.2: Parallel, specification-based testing tool.

Figure 16.3: Program-based testing tool.
16.1 Related Work

Using a programming language as a specification language is not a new idea. In a technical [68] report by North, he compares test data generation from specifications written in three different languages all modeling the so-called triangle problem. The languages used were VDM-SL, Miranda, and Prolog, where the latter two are not specification languages.

He finds Prolog unsuitable as a specification language (at least for the triangle problem), since it is not a typed language, and thus it does not have important data structures such as sets. These all have to be implemented in Prolog.

Miranda does have some sort of typing system, but it is not as flexible as the one in VDM-SL. However, Miranda is not as abstract as VDM-SL, e.g. it does not have sets. Sets can be implemented as an abstract data type, but this result in a less favorable syntax compared to VDM-SL. Furthermore, Miranda does not support universal (∀) or existential (∃) quantifiers, as in the case of VDM-SL, but rather has to implement this through some iteration sequence, which makes an expression more complex to analyze.

North’s conclusions are true for all other programming languages. They all lack important abstractions such as sets and set operations.

The technique used in this work to find anomalies between subjects is essentially a type of output spectra comparison. It is straightforward, however, to extend this work to use other types of spectra such as syntactic spectra or semantic spectra [66,67,69].

This thesis studies the coupling between branch coverage and defect coverage (mutation coverage), to see whether branch coverage is a reliable instrument to use when tuning automatic input generators. Another approach would be to create reliability-growth models to relate, for example, branch coverage to defect coverage [70].
Chapter 17
Basic Concepts

— I have only one yardstick by which I test every major problem - and that yardstick is: Is it good for America?

Dwight D. Eisenhower

A program $p$ is a function from input to output, $p : Input \rightarrow Output$. Input encapsulates the program state before execution, that is, it contains all variables used or defined by the program, mapped to their original values. Thus, input not only contains program input parameters but all information needed to specify program behavior such as referenced global variables. Similarly, output represents the state after execution as well as the program return value, if any. This means that input parameter values after execution and side effects such as console output can be found here.

A specification $s$ of a program $p$ is a function $s : Input \times Output \rightarrow \{false, true\}$ that verifies whether the actual output of $p(i)$ for some input $i$ is equal to the expected output of $p(i)$. The first parameter of the specification is the input and the second is the actual output.

Practically, this means that a program is something that takes input and results in output, while a specification is something that can decide whether the output is correctly derived from the input. In Listing 17.1, for example, we see a specification and a program that calculates the integer square root of its argument. Note the how the difference in how they express the original problem. Specifications are generally given in terms of what to do, while programs most often state how to do it.

17.1 Goal of Testing

Despite the structural difference of the two, a specification $s$ and program $p$, are only different ways of defining the same problem $q$. While $s$ and $p$
Listing 17.1: A program that calculates the integer square root of its argument.

```c
int sqrt(int x) {
    int square = 1;
    int delta = 3;
    while (square <= x) {
        square += delta;
        delta += 2;
    }
    return (delta/2 - 1);
}
```

```c
boolean sqrt_spec(int x, int y) {
    return y^2 <= x && x <= (y+1)^2;
}
```

Figure 17.1: The possible configurations of two problem descriptions $a$ and $b$.

are given in some formal notation, $q$ is only described informally, perhaps only as an idea in the head of a programmer.

Given two problem definitions, or simply problems, $a$ and $b$, we loosely define the following operators (also see figure 17.1).

$a = b$  We say that $a$ is equal to $b$, when all functionality in $a$ is also in $b$ and vice versa.

$a \supset b$  We say that $a$ contains $b$ or $b$ is contained by $a$, when all functionality in $b$ is also in $a$ but not vice versa.

$a \not= b$  We say that $a$ is not equal to $b$, when some functionality of $a$ is not in $b$ or vice versa. Note that $a \supset b$ implies that $a \not= b$.

It is important to note that the goal of software testing is not to show that $s = p$. In general, it is undecidable to do that. Even if we do show
s \neq p \text{ we still must show that either } s = q \text{ or } p = q \text{ to be sure that the original problem has been correctly implemented. However, that step cannot be automated since } q \text{ is informal. Instead, we show the opposite: } s \neq p. \text{ It is simple to prove that } s \neq p \text{ implies that at least one of } s \text{ and } p \text{ contains an error. Thus, we have }

\[ s \neq p \implies p \neq q \lor s \neq q. \]

If } s \neq p \text{ cannot be shown we conclude by saying that } s \neq p \text{ cannot be decided.}

### 17.2 Coverage Criteria

The coverage criterion (or adequacy criterion) plays one of the most important roles. A coverage criterion is used to determine the areas or properties of its subject that have been or need to be tested. The subject, that is, the element on which coverage is measured, is often related to the program being tested or its specification. For example, in statement coverage it is required that all statements of the program have been executed.

It is important to note that for the best result coverage must be measured both on the specification and on the program, because only the functionality expressed in the subjects of the coverage criteria is explored. This means, for example, that tests generated from a program-based criterion can only test functionality expressed in the program. If the program is, for some reason, missing functionality (with respect to its specification) this cannot be detected. In the same way, tests generated from a specification-based criterion can only test functionality expressed in the specification.

Coverage criteria is language dependent. Languages with e.g. control-flow and data-flow can use criteria based on the control-flow and data-flow graph. Other languages, such as the specification language of VDM-SL, do not always have a clear control-flow nor data-flow, and must therefore define their own criteria.

A coverage monitor is a tool used to monitor coverage in run-time. Probes, that is, counters, are inserted at certain positions in the subject (a specification or program) according to the coverage criterion in use. Understandably, each language and each coverage criterion requires its own implementation of the monitor. In listing 17.2 we see the `contains` function on page 140 instrumented for branch coverage.
Listing 17.2: A function instrumented for branch coverage.

```java
boolean contains(Object[] a, Object r) {
    for (int i = 0; i < a.length; i++) {
        probe[0]++;
        if (r == a[i]) {
            probe[1]++;
            return true;
        } else {
            probe[2]++;
        }
    }
    probe[3]++;
    return false;
}
```

17.3 Input Generation

Let us review the material in chapter 2 regarding input generation. According to one of the most fundamental principles of testing, called partition testing, a coverage criterion divides the input space of the subject into subdomains, among which one or more representatives are selected as inputs. The rationale behind this is that each representative tests its subdomain equally well.

The strength of the generated input set, that is, its effectiveness at detecting faults, depends on how well the criterion partitions the input domain. The stronger a criterion is, the more details of the subject are taken into account. In general this means that more inputs are needed to meet the criteria, which implies more work. In this way coverage criteria play a central role in input generation, since it ultimately decides what is to be tested and how thoroughly it will be tested.

Basically, there are two ways of selecting inputs from the subdomains: systematically or randomly. Systematic input generators work in two steps. Firstly, the input space is partitioned into subdomains using a coverage criterion. Each subdomain can be described by a constraint, formulated on the input variables of the subject, that only holds for inputs of that subdomain. The goal of partitioning is to identify these constraints. Earlier approaches described in the literature applied complex program analysis such as symbolic execution and linear approximation of program decisions.

Secondly, the systematic generator seeks a solution to each constraint
because a solution directly corresponds to an input from the subdomain associated with the constraint. This is the actual input generation step. Techniques studied in the literature have used optimization techniques such as mixed integer programming, genetic algorithms, and constraint programming.

The second kind, which is called random input generator, is actually an extreme version of the first. Here the inputs are generated randomly over the complete input space. In other words, a random generator is an extreme version of systematic generator in the sense that the partitioning process only results in one large subdomain (the complete input space).

Theoretically, this means that systematic testing is statistically as good as random testing only if subdomains are sampled in proportion to their size. A difficulty here is how to estimate subdomain size.

Random generators automatically distribute inputs according to subdomain size by common laws of statistics. However, particularly small subdomains can pose a problem since the random tester may never find them. Naturally, one could apply a two-step procedure here by systematic testing of those subdomains that were left untested after the initial random test.
Chapter 18

Specification-based Testing Tool

— A small error in the former will produce an enormous error in the latter.

Jules H. Poincare

A model of a testing tool is given in figure 16.1. This particular testing tool is specification-based, meaning that a specification is used to verify the actual program result. To contrast this we can imagine another type of testing tool that verifies the actual program result by executing a reference program. Such a tool is referred to as being program-based (see figure 16.3).

Thus, the term specification-based testing tool refers to the architecture of the testing tool and not how it generates tests. Thus there is no contradiction in a specification-based testing tool being equipped with a program-based input generator.

This chapter presents the results of a study of a prototype implementation of a specification-based testing tool. Listing 18.1 presents a simplified version of the top loop of the tool. As can be seen, in each iteration the program is executed on a new input (line 8). The result is verified against a specification (line 12). Two coverage monitors are used to keep track of program and specification coverage (lines 7 and 11).

Naturally, it is valuable to have a detailed test report containing information about all tests and the accumulated coverage achieved by each successive test. This information is useful when doing post-test analysis, for example, when deciding whether more tests should be generated and if generation strategy must be modified. Another benefit is that the next time the same subjects (program and specification) are tested, the tool could start by executing the saved inputs of the previous test before consulting the input generator.

Naturally, all executed tests cannot be added to the report. A random input generator, for example, may need to generate thousands of input
Chapter 18 Specification-based Testing Tool

Program: experiment.triangle.TriangleProgram_0
Specification: experiment.triangle.TriangleSpecification_0
Passed: 7
Failed: 0

Index: Prog( T F) Spec( T F)
0: 13%( 13% 0%) 20%( 20% 0%) [] --> Exception ==> true
4: 44%( 44% 0%) 60%( 60% 0%) [1, 2, 4] --> Exception ==> true
5: 75%( 75% 0%) 80%( 80% 0%) [8, 5, 8] --> ISOSCELES ==> true
6: 81%( 81% 0%) 80%( 80% 0%) [-1, 5, 2] --> Exception ==> true
9: 88%( 88% 0%) 80%( 80% 0%) [7, 6, -2147483648] --> Exception ==> true
27: 94%( 94% 0%) 87%( 87% 0%) [3, 6, 7] --> SCALENE ==> true
576: 100%(100% 0%) 93%( 93% 0%) [8, 8, 8] --> EQUILATERAL ==> true

Program: experiment.triangle.TriangleProgram_8
Specification: experiment.triangle.TriangleSpecification_0
Passed: 2
Failed: 1

Index: Prog( T F) Spec( T F)
0: 13%( 13% 0%) 20%( 20% 0%) [] --> Exception ==> true
4: 50%( 50% 0%) 53%( 53% 0%) [9, -2147483648, -1] --> Exception ==> true

Index: Prog( T F) Spec( T F)
5: 56%( 50% 44%) 80%( 53% 67%) [5, 8, 8] --> Exception ==> false

Figure 18.1: A sample output from the testing tool using a random input generator.

before reaching an acceptable level of coverage. Therefore, some type of filtering of tests must be performed. This is done in the final step of the loop. Here it is decided whether the generated tests should be added to the test report.

The filtering mechanism used by the prototype is based on the accumulated coverage and the outcome of the test. The test report keeps separate coverage records of each subject. Each record is divided into two categories: passing or failing coverage. Passing program coverage, for instance, is the accumulated program coverage of all passing tests that have been added to the record so far. If a passing test improves either the passing specification or program coverage, it is saved (added to the record). Likewise, if a failing test improves either the failing specification coverage or the failing program coverage, it is saved.

Figure 18.1 shows the reports of two tests performed using a random input generator. In the first test, between TriangleProgram_0 (TP0) and TriangleSpecification_0 (TS0), the tool found 7 (improving) passing tests, and no failing tests.

The index in front of each test represents the order number of the test starting at 0, meaning that after 6 tests (5+1) the tool found an anomaly between TP8 and TS0.
18.1 Control-flow Based Coverage Criteria For Specifications

Listing 18.1: A testing tool with a serial verifier.

```java
Report test(Spec spec, Prog prog, InputGenerator inputs) {
    Report report = new Report();
    while (inputs.hasNext() && report.numberOfFailingTests() <= LIMIT) {
        Input input = inputs.next();

        CoverageMonitor progMon = new CoverageMonitor(prog);
        Output output = prog.invoke(input.clone());
        progMon.deactivate();

        CoverageMonitor specMon = new CoverageMonitor(spec);
        boolean outcome = spec.invoke(input, output);
        specMon.deactivate();

        if (report.isImprovingTest(progMon, specMon, outcome)) {
            report.addTest(input, output, outcome, progMon, specMon);
        }
    }
    return report;
}
```

The testing tool stops once the generator is out of inputs or the number of failing tests has reached a certain limit. In this work this limit has been set to 0, meaning that the tool stops as soon as it finds the first failing test.

18.1 Control-flow Based Coverage Criteria For Specifications

In this work it is studied whether it is possible to use control-flow based coverage criteria for specifications. A problem with control-flow based criteria, however, is that full coverage of a specification is generally not possible.

Many criteria, such as control-flow based criteria, are constructed to force execution of all parts of the subject. For programs this property is desirable, but for specifications it is not. To understand why, consider the hypothetical function `boolean foo(boolean a)` with a specification `a ∨ b`, where `b` is the return value.
Table 18.1: The subdomains generated from $a \lor b$ using Meudec and MCDC.

| Meudec | $a$ | $b$ | $a \lor b$ | MCDC | $a$ | $b$ | $a \lor b$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \land b$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$\neg a \land \neg b$</td>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$\neg a \land b$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$\neg a \lor b$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$a \land \neg b$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$a$</td>
<td>$T$</td>
<td>$F/T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Table 18.1 shows the subdomains, and their solutions, generated from the expression using Meudec’s criteria and MCDC [71]. As can be seen, each results in three subdomains. However, while solutions to Meudec’s subdomains result in the expression being true, one of the MCDC subdomains results in false.

A solution to a specification subdomain is a pair $(i, o)$ where $i$ is the program input and $o$ is the expected program output. The pair is referred to as solution, test or specification input.

A true test $(i, o)$, or a desired test, is an input $(i, o)$ to a specification $s$ resulting in $s(i, o)$ being true. True tests exercise the desired behavior of the program, that is, the functionality that the specification requires from the program.

On the contrary, a false test, or an undesired test, is an input to a specification resulting in it being false, meaning that it exercises the undesired behavior of the program.

False tests are not good tests. For example, consider MCDC over a specification $foo$ $\text{spec}(a, b) = a \lor b$, and the program $b = \text{foo}(a)$, where $a$ and $b$ are booleans. As can be seen in 18.1 $(F, F)$ is a false test because the specification evaluates to false. Literally the test says that the expected result of $\text{foo}(F)$ is not $F$. But that is not a very strong test when it comes to determining the functionality of $\text{foo}$.

A better test would be a solution to the second subdomain, $(F, T)$, which is a true test, meaning therefore that the expected result of $\text{foo}(F)$ is $T$.

To give a more elaborate example of false tests, consider the specification of a sort program $\text{void sort(int}[] \text{a)}$ given in listing 18.2. Branch coverage of $\text{sort\_spec}$, for example, requires that line 5 is executed. Because the return value is constantly false we know that all solutions to this subdomain are false. Such a domain is called a false domain.

A test covering line 5 would be, for example, ($[8, 2, 7]$, [44, -2]). Running this test is the same as verifying that the expected result of $\text{sort([8, 2, 7])}$ is not equal to [44, -2]. But there are lots of things
that the expected result of `sort` is not: it is not a string, not a double, it is not an orange etc. This is exactly why true tests are much stronger than false ones. It is better to test things things that are, for example, that the expected result of `sort([8, 2, 7])` is equal to `[2, 7, 8]`.

A consequence is that it is possible to achieve full coverage using a set of only false tests. For example, the tests `([], [44, -2])`, `([], [-2, 44])`, `([2], [-2, 44])` achieve 100% branch coverage, but they are not good tests.

Therefore, it is important when generating tests from a specification using a program-based coverage criteria that false tests are not included. Or actually, a false test can be included as long as it does not prevent a true test from being included. Practically, this is rather difficult to determine. Therefore not including false tests at all is a simple solution.

Unfortunately, this means that reaching full specification coverage is impossible. In the sort example, coverage of line 5 is prohibited because it would result in a false test. The effect is that it is difficult to assess the strength of a set of tests by only looking at the degree of coverage. A good set of inputs for sort specification has perhaps 83% coverage, while
the bad set above has 100%.

To this there are two important questions. The first is whether there is a simple way to filter out only the true subdomains such that full coverage can be determined. In the general case there is not. The problem amounts to showing that a particular subdomain has no solution. However, there is specific case, which occurs rather frequently throughout a specification, and that is when a literal `true` or `false` is returned. With this information it is possible to statically remove at least some of the infeasible subdomains.

The second question is whether a program-based coverage criteria becomes too weak when applied to specifications. This is one of the questions pursued in the experiment presented in following sections.

## 18.2 Method

Mutation coverage is used to evaluate the fault-detection effectiveness of an input set or a testing tool. Unlike control-flow based criteria, for example, mutation coverage is not based on the test subject itself. Instead faulty versions of the subject, called mutants, are created by introducing faults, so-called mutations, into the subject code.

An example would be to replace a loop-variable reference in the body of a `for` loop can be replaced by a constant value or a reference to another variable. Another mutation could be to add a logical negation to the condition of an `if` statement. The idea is that the errors introduced (and their distribution) should correspond as much as possible to real errors.

The goal of mutation coverage is to generate a set of inputs such that all mutants are detected by at least one input in the set. A mutant is detected by an input if the mutant produces a different output than the original subject for that particular input.

Below is a list of fault classes used in this work.

**Operator Fault** An operator of an expression is changed (negation is not included here). Examples: `≠` is replaced by `=`, `∨` by `∧`, or `≥` by `<.

**Negation Fault** An expression is negated. Example: `a < b` is replaced by `a ≥ b`.

**Order Fault** The order of an operation or a sequence of statements is changed.

**Deletion Fault** A block of code or expression is omitted.
Exception Fault  A thrown exception is of the wrong type or has the
wrong message. Not applicable to specifications since these are not
allowed to throw any exceptions (must not return anything else but
true or false).

Reference Fault  An expression, variable, or constant is replaced by an-
other expression, variable, or constant. For example, \( i \) is replaced
by 1, 0, \( i - 1 \), or \( j \).

Initialization Fault  A variable is set to a constant or random value just
before first use.

Because specifications are implemented in a programming language it
is a sound assumption that naturally introduced faults of a specification
can be categorized into the same classes as the faults of programs. What
might differ is the distribution of the faults among the classes.

The term true mutant is used to denote mutants that are semantically
different from the original. For all inputs at least one results in a different
output than the original. A false mutant, on the other hand, is semanti-
cally equal to the original, meaning that they will have the same output
as the original for all inputs.

Based on this, the fault-detection effectiveness (FDE) of input set \( X \) is
defined as the number of mutants that it identifies. The more mutants
identified, the higher the FDE. Formally, let \( a_0 \) be the subject being tested
and let \( A = \{a_1, \ldots, a_n\} \) be mutant versions of \( a_0 \), preferably \( a_0 \neq a_i \) for
all \( a_i \in A \). Then the FDE of an input set \( X \) is defined as the number of
mutants in \( \{a_i \mid a_i \in A \land x \in X \land a_0(x) \neq a_i(x)\} \).

This method has been used by in a study by Schroeder et al. when
comparing the FDE of N-way and random test suites [72].

In specification-program testing, in contrast to program testing or spec-
ification testing, the subject of the test is a specification-program pair
\((s, p)\). Accordingly, the set of mutants consists of SP-pairs and identifying
a pair \((s, p)\) as a mutant corresponds to showing that \( s \neq p \), that is,
\( \neg s(i, p(i)) \) for input \( i \).

18.3 Test Subjects

The test subjects (mutants) used in this experiment are based on a slightly
modified version of Myers’ [73] Triangle problem. In this version the sides
are constrained to integers, rather than objects. Moreover, the modified
problem specifies what to do when an invalid triangle is found.
Table 18.2: Fault distribution in the Triangle Experiment.

<table>
<thead>
<tr>
<th>Class</th>
<th>Spec</th>
<th>Prog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator Fault</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Negation Fault</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Order Fault</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Deletion Fault</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Exception Fault</td>
<td>N/A</td>
<td>4</td>
</tr>
<tr>
<td>Reference Fault</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Initialization Fault</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total faults</td>
<td>15</td>
<td>24</td>
</tr>
</tbody>
</table>

**Definition 18.1 (Triangle Problem).** The program is passed an array of integers containing three values interpreted as representing the length of the sides of a triangle. The program returns whether the triangle is \textsc{SCALENE}, \textsc{ISOSCELES}, \textsc{EQUILATERAL}. If the sides does not represent a triangle an \texttt{IllegalArgumentException} is thrown.

Despite its small size, the triangle problem has a fairly complex logic. Myers made pedagogical use of this fact when showing that the problem needed many inputs to be tested sufficiently. He lists 14 criteria a good input set should satisfy.

The triangle problem has been used in several other research papers on automatic testing. In fact, two input sets found in the literature are used as a reference to the random inputs used in the experiment.

From a correct implementation of the triangle specification and the triangle program, called \(s_0\) and \(p_0\), a set of mutant specification \(S = \{s_0, s_1, \ldots s_{15}\}\) and mutant programs \(P = \{p_0, p_1, \ldots, p_{24}\}\) have been created according to the fault distribution presented in table 18.2. Only one fault in each mutant was injected. The originals \(s_0\) and \(p_0\) are the only false mutants that are members of the sets.

The set of \(sp\)-mutants is defined as the Cartesian product \(S \times P\) resulting in a set of 400 pairs. Naturally at least one of the pairs is a false mutant, namely \((s_0, p_0)\), but there may be more.

### 18.4 Test Inputs

The objective of the experiment is to study the FDE of the testing tool when using branch coverage. The variable of the experiment is the input generation technique.
For this reason, three random input generators, R1, R2, and R3, have been constructed. Even though their mutual strength (FDE) was unknown before the time of the experiments, they were constructed with increasing strength in mind.

**R1** This input generator produces \( n \)-sided polygons \([s_1, \ldots, s_n]\) with side lengths \( s_i \), where \( n \) is random number \( 0 \leq n \leq 4 \), and where each side \( s_i \in \{m, \ldots, M\} \), where \( m \) and \( M \) are the minimum and maximum Java integers.

**R2** This input generator works in the same way as R1, however, the sides are randomly generated from a much smaller domain \( s_i \in \{m, m + 1, -1 \ldots, 9, M - 1, M\} \).

**R3** This input generator is the result of reducing R2 even further. It consists of one random \( n \)-sided polygon for \( n \in \{0, 1, 2, 4\} \). The rest of the polygons in the input set are three-sided. That is, the first 4 polygons of this set have 0, 1, 2, and 4 sides respectively and the rest have 3 sides. All sides are drawn from the same domain as the sides in R2.

The term input generator is often used interchangeably with input set. A random input generator can be viewed as a dynamic set of inputs that is regenerated every time it is enumerated.

As a reference to the random input generators, a number of (static) input sets are used. The first reference set, which is referred to as North’s Inputs (NI) is based on a hand-generated set of inputs and is believed to be very strong. The second, called Meudec’s Inputs (MI) is a machine-generated set of inputs generated by Meudec [9] in his thesis on automatic test generation. The strength of this input set is unknown. The final sets of inputs are called Student Inputs (SI), which is a collection of 13 input sets generated by undergraduate students.

Table 18.3 gives a list of test cases hand-generated by North [68] from Myers’ original specification using various black-box techniques: equivalence partitioning, boundary value analysis, cause-effect graphing, and error guessing. North’s test cases conform to Myers’ list of suggestions for what good test sets for the triangle problem should cover.

Test cases marked by a dagger (\( \dagger \)) either fail on compilation or are not meaningful in other ways. For instance, test case 17 does compile with our Java implementation. However, Java, which is the language used by the prototype, automatically casts the char ‘A’ to its unicode value 65, thus the test case is no longer meaningful. Alternatively, the character
Table 18.3: North’s inputs for the triangle problem, where M is the highest integer.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [0, 0, 0]</td>
<td>INVALID</td>
</tr>
<tr>
<td>2 [0, 1, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>3 [1, 0, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>4 [1, 1, 0]</td>
<td>INVALID</td>
</tr>
<tr>
<td>5 [3, 1, 2]</td>
<td>INVALID</td>
</tr>
<tr>
<td>6 [1, 3, 2]</td>
<td>INVALID</td>
</tr>
<tr>
<td>7 [2, 1, 3]</td>
<td>INVALID</td>
</tr>
<tr>
<td>8 [1, 2, 5]</td>
<td>INVALID</td>
</tr>
<tr>
<td>9 [5, 2, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>10 [2, 5, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>11 [5, 1, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>12 [1, 5, 1]</td>
<td>INVALID</td>
</tr>
<tr>
<td>13 [1, 1, 5]</td>
<td>INVALID</td>
</tr>
<tr>
<td>14 [1, 2, -6]</td>
<td>INVALID</td>
</tr>
<tr>
<td>15 [-2, -2, -2]</td>
<td>INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>16 [2, 2.3, 2]</td>
<td>INVALID</td>
</tr>
<tr>
<td>17 ['A', 2, 3]</td>
<td>INVALID</td>
</tr>
<tr>
<td>18 ['A', 'A', 'A']</td>
<td>INVALID</td>
</tr>
<tr>
<td>19 [2, 3]</td>
<td>INVALID</td>
</tr>
<tr>
<td>20 [4, 4, 4, 4]</td>
<td>INVALID</td>
</tr>
<tr>
<td>21 [M, M, 1]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>22 [M, M, M]</td>
<td>EQUILATERAL</td>
</tr>
<tr>
<td>23 [M + 1, M - 1, M]</td>
<td>INVALID</td>
</tr>
<tr>
<td>24 [1, 1, 1]</td>
<td>EQUILATERAL</td>
</tr>
<tr>
<td>25 [1, 2, 2]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>26 [2, 1, 2]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>27 [2, 2, 1]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>28 [3, 2, 2]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>29 [2, 3, 2]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>30 [2, 2, 3]</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>31 [2, 3, 4]</td>
<td>SCALENE</td>
</tr>
<tr>
<td>32 [3, 2, 4]</td>
<td>SCALENE</td>
</tr>
<tr>
<td>33 [3, 4, 2]</td>
<td>SCALENE</td>
</tr>
<tr>
<td>34 [4, 3, 2]</td>
<td>SCALENE</td>
</tr>
<tr>
<td>35 [4, 2, 3]</td>
<td>SCALENE</td>
</tr>
<tr>
<td>36 [2, 4, 3]</td>
<td>SCALENE</td>
</tr>
</tbody>
</table>
Table 18.4: Meudec’s inputs for the triangle problem.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 2</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 3</td>
<td>INVALID</td>
</tr>
<tr>
<td>4</td>
<td>INVALID</td>
</tr>
<tr>
<td>5</td>
<td>EQUILATERAL</td>
</tr>
<tr>
<td>6</td>
<td>EQUILATERAL</td>
</tr>
<tr>
<td>7</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>8</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>9</td>
<td>SCALENE</td>
</tr>
<tr>
<td>10</td>
<td>SCALENE</td>
</tr>
<tr>
<td>11</td>
<td>INVALID</td>
</tr>
<tr>
<td>12</td>
<td>INVALID</td>
</tr>
<tr>
<td>13</td>
<td>INVALID</td>
</tr>
<tr>
<td>14</td>
<td>INVALID</td>
</tr>
<tr>
<td>15</td>
<td>INVALID</td>
</tr>
<tr>
<td>16</td>
<td>INVALID</td>
</tr>
<tr>
<td>17</td>
<td>INVALID</td>
</tr>
<tr>
<td>18</td>
<td>INVALID</td>
</tr>
<tr>
<td>19</td>
<td>INVALID</td>
</tr>
<tr>
<td>20</td>
<td>INVALID</td>
</tr>
<tr>
<td>21</td>
<td>INVALID</td>
</tr>
<tr>
<td>22</td>
<td>INVALID</td>
</tr>
<tr>
<td>23</td>
<td>INVALID</td>
</tr>
<tr>
<td>24</td>
<td>INVALID</td>
</tr>
<tr>
<td>25</td>
<td>INVALID</td>
</tr>
<tr>
<td>26</td>
<td>INVALID</td>
</tr>
<tr>
<td>27</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 28</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 29</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 30</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 31</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 32</td>
<td>INVALID</td>
</tr>
<tr>
<td>33</td>
<td>INVALID</td>
</tr>
<tr>
<td>† 34</td>
<td>INVALID</td>
</tr>
</tbody>
</table>
could be entered as a String "A", but this would make the compiler signal an error.

In table 18.4 a list of test cases generated by Meudec is shown. Meudec concludes his thesis by using his method to generate test cases for the triangle problem. His list of tests is presented in table 18.4. The symbol * stands for the undefined value in VDM-SL and does not have a representation in Java. Test cases containing this symbol are therefore omitted.

Note that Meudec generated his set using an incorrect minimization algorithm (see section 9.3). It is not clear whether this has affected the outcome of the set or not. If applied correctly, his algorithm can work. However, it can also be fooled to produce non-minimal input sets.

It is important to point out that the outcome of a test of an $sp$-pair is determined by the specification and not by the expected output given in the tables of North and Meudec. The expected outputs of the tables are provided only for the purpose of completeness and to show the intention of the test.

The Student Inputs ($si$) is a collection of 13 input sets that were generated by fourth-year computer-science students. The inputs were generated as an exercise in a course on software testing. The actual input generation was not performed in a controlled environment, meaning that the students could have been helped by others when generated the inputs. The students were given clear instructions and they were encouraged to use Meyers 14 criteria for good triangle tests.

All reference input sets were derived from the correct specification $s_0$, and therefore, it may that they do better in detecting mutants of the form $(s_0, p_j)$, but worse for mutants $(s_i, p_j)$ where $i > 0$. However, good tests should not depend on the specification being correct in order to show that $s \neq p$.

## 18.5 Results

Each input set was executed against the full set of 400 $sp$-mutants ($S \times P$). While $NI$ and $MI$ are backed up by fixed sets, which in turn give the same results on each execution, the random generators have slightly different results on each execution. Therefore, the results of the random generators are calculated as an average over a number of executions. The random generators were set to generate a maximum of 10000 inputs and were sampled at selected intervals. In this way it is possible to plot the results as a function of the number of inputs.

The test results from the execution of the random generators are given
Figure 18.2: $S \times P$: 400 mutants, 3 undetectable

Table 18.5: Random generators executed against $S \times P$: 400 mutants, 3 undetectable

<table>
<thead>
<tr>
<th>Inputs</th>
<th>R1</th>
<th>std</th>
<th>R2</th>
<th>std</th>
<th>R3</th>
<th>std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>273</td>
<td>7.0</td>
<td>269</td>
<td>12</td>
<td>253</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>226</td>
<td>9.7</td>
<td>222</td>
<td>3.8</td>
<td>253</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>167</td>
<td>6.7</td>
<td>154</td>
<td>7.0</td>
<td>146</td>
<td>8.3</td>
</tr>
<tr>
<td>10</td>
<td>126</td>
<td>5.5</td>
<td>103</td>
<td>8.0</td>
<td>54</td>
<td>3.2</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>4.7</td>
<td>58</td>
<td>1.6</td>
<td>21</td>
<td>2.9</td>
</tr>
<tr>
<td>50</td>
<td>49</td>
<td>2.7</td>
<td>33</td>
<td>2.3</td>
<td>11</td>
<td>1.2</td>
</tr>
<tr>
<td>100</td>
<td>33</td>
<td>3.1</td>
<td>21</td>
<td>2.7</td>
<td>6.3</td>
<td>0.8</td>
</tr>
<tr>
<td>250</td>
<td>23</td>
<td>0.82</td>
<td>9.2</td>
<td>2.2</td>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>500</td>
<td>22</td>
<td>0</td>
<td>5.3</td>
<td>1.4</td>
<td>3.5</td>
<td>0.6</td>
</tr>
<tr>
<td>1000</td>
<td>22</td>
<td>0</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>22</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 18.3: Coverage of North Inputs.

Figure 18.4: Coverage of Meudec Inputs.

Figure 18.5: Coverage of R1.

Figure 18.6: Coverage of R2.

Figure 18.7: Coverage of R3.
18.6 Discussion

in table 18.5 as the number of undetected mutants at a certain input. For instance, after 10 inputs R3 has 54 undetected mutants. After 1000 inputs it only has 3 mutants left. The lower the value of the number of undetected mutants the better.

Figure 18.2 shows a plot of the number of undetected mutants as a function of the number of inputs. Here, the result of NI (5 undetected) and MI (28 undetected) are plotted as two horizontal lines. Note that the y-axis is logarithmically scaled.

The average score of SI was 44. The best score was 31 and the worst 74. The latter student possibly misunderstood the exercise and generated inputs for branch coverage only.

18.6 Discussion

Surprisingly, our initial assumption of that NI would be the strongest of all sets proved to be wrong. NI leaves 5 mutants undetected and MI leaves as many as 28. In fact, R3 performs better than NI after only 160 inputs on average. R2 also performs better than NI but only after around 600 inputs.

It can be seen from the low standard deviations that the results of the random generator are very stable. Naturally the deviation decreases as more inputs are generated.

The three mutants left by R3 demonstrates a limitation of the tool. The best scenario would be that the mutants all have \( s = p \). However, after a closer analysis it can be determined that this is only true for one of the mutants. The other two have \( s \supset p \).

The reason these two mutants were missed is not that the inputs sets are weak. In fact, the design of the tool (see figure 16.1) makes it impossible to detect the situation where \( s \supset p \), though detecting \( s \subset p \) works fine. The problem is that the serial design keeps the tool from exploring the input space of the specification independently of the exploration of the program’s input space. By serial it is meant that the tool first executes the program and then verifies the result, meaning that the output parameter of the specification is calculated by the program and not by an independent input generator trying to maximize coverage. In this way the exploration of the input space of the specification is limited by the possible outputs of the program.

Alternatively, the parallel layout given in figure 16.2 can be used. Here no parameters are constrained. However, parallel specification-based testing tools have much worse performance than serial tools when random
input generators are used. This is because the output parameter of the
specification must be randomly generated, instead of calculated by the
program. Additionally, the outcome of the specification must be true for
the given input and output.

Figures 18.3–18.7 plot the coverage as a function of the number of
inputs. Coverage for programs and specifications is measured in terms of
percentage of branch coverage.

The figures also plot the adjusted mutation coverage, that is, the ratio
between the number of detected mutants and the number of mutants it
is possible to detect using the tool (397 mutants in this case). Note that
the x-axis is logarithmically scaled.

As expected, the specification coverage is asymptotically lower than
the program coverage. This is a result of the serial design of the tool and
the false subdomains explained in section 18.1.

R3 performs better than R1. However, what is interesting, despite quite
low program and specification coverage, with 84.5% and 77.3%, respec-
tively, R1 still identifies quite a large proportion of the faults (95.2%).
Compare this to the results of M1 which has a program coverage of 90.4%
and a specification coverage of 88.6% but only detects 93.7% of the faults.

What is more, for all random generators, the mutation coverage is
always higher than the coverage of the program and the specification. The
detection rate (of mutants) levels out before the branch coverage does,
indicating that no more faults can be found with the input generator in
use. Additionally, the number of detected faults can be expected to be of
at least the same proportion.

This reasoning seems to be true for M1 as well. However, the detection
rate of M1 levels out just after the coverage. For N1, on the other hand,
the ratio of detected mutants is slightly less than the program coverage,
99.5% compared to 100%.

Thus, when using random input generation it seems that program
branch coverage could be used as an indication of the level of defect
coverage. Low program coverage that has leveled out indicates that a
better input generator is needed.

The false subdomains encountered in specification branch coverage does
not seem to be a problem when it comes to random testing. The coverage
of the specification follows the program coverage well, though, it is lower. Therefore, specification coverage can be used to further strengthen the stopping criteria.
Yet another solution to the oracle problem is to use a reference program to verify the program output, instead of a specification. In this way the false subdomains of the specification are not a problem.

As can be seen in figure 16.3 such a tool is inherently parallel in its design, making it possible to also detect if \( r \supset p \), where \( r \) is the reference program and \( p \) is the program under test.

The first impression of this tool is the additional cost of implementing the reference program; two programs instead of one. However, in many situations there naturally exist two or more versions of a program. The reference program could be a prototype, or a version of the program compiled for a different platform. An example from the academic world is the teacher’s solution to a programming exercise corresponding to the reference and the students’ solutions being the programs under test. The reference program can also be a mutated versions of the program under test.

19.1 Method

Mutation testing is used also to to evaluate the reference-based testing tool. This time the mutants consists of pairs \((r, p)\) of a reference program \(r\) and a program \(p\). This method has been used in study on program spectra by Harrold et al [69].
Chapter 19 Program-based Testing Tool

The set of mutant pairs $M$ is constructed from a set of program mutants $P$ such that each combination of programs $(r, p)$ or $(p, r)$ is represented exactly once.

$$M = \{(r, p) \mid r, p \in P \land r \neq p \land (p, r) \notin M\}$$

Intuitively, if a $P \times P$ is written as a multiplication table, then $M$ corresponds to the lower (or higher) triangle of the table. The size of $M$ is $|M| = \frac{|P|(|P| - 1)}{2}$. For the triangle problem this results in a set of 325 triangle-program pairs.

In addition to the triangle problem two other problems are explored: Sort Problem, and Calculator Problem.

**Definition 19.1 (Sort Problem).** Sorting an array of integers in ascending order.

The mutants of sort problem are based on two different implementations. The first is a Bubble-Sort implementation with a total of 25 mutants including the original. The second is a Quick-Sort implementation with a total of 17 mutants. Altogether this results in 861 sort pairs.

The random input generator for the sort problem generates lists of length 20 with elements $e \in \{m, m - 1, -5, 100, M - 1, M\}$.

**Definition 19.2 (Calculator Problem).** Calculate the value of an infix expression consisting of variables, decimal numbers, operators (assignment, $+, -, \cdot, /$) and parentheses.

Of the three problems, the calculator problem is the most complex. The original implementation consists of about 300 lines of code.

The set of mutants for the calculator problem consists of 40 mutant program, including the original. About half of these are based on real errors actually made when implementing the original.

The input generator for the calculator generates expression trees with a height between 1 and 10. The leafs of a tree are randomly assigned random variable names or random decimal numbers. The internal nodes are randomly assigned to operators. The whole tree is then recursively transformed to an infix expression. Between each node 0–2 spaces are inserted. Finally, from the resulting infix string a random non-digit character is deleted. Doing so possibly creates an incorrect infix expression such that error recovery of the calculator is tested as well.
19.1 Method

Figure 19.1: Coverage of triangle mutants.

Figure 19.2: Coverage of sort mutants.

Figure 19.3: Coverage of calculator mutants.
19.2 Results

Figure 19.1 shows the adjusted mutation coverage of the triangle pairs as well as the average branch coverage of all programs (and reference programs). Of the 325 triangle pairs, the testing tool managed to fail all but three.

A manual inspection of the three undetected pairs showed that all had in fact \( r = p \), thus 100% adjusted mutation coverage was reached.

Similarly, figures 19.2 and 19.3 shows results of the sort test and the calculator test. The tool failed all but 7 of the 861 sort pairs and all but 2 of 780 calculator pairs.

A manual inspection showed that all pairs had \( r \neq p \), meaning that 100% adjusted mutation coverage was reached.

19.3 Discussion

Similar observations and conclusions can be made in this experiment to those made in the experiment on the specification-based tool. The mutation coverage is always higher the program (branch) coverage and it levels out earlier and faster than the program coverage.

Using reference programs as oracles does not seem to decrease the fault-detection effectiveness. The reference program can be implemented very differently, compared to the program subjected to the test, and still perform well. A good example is the sort problem where Bubble Sort and Quick Sort where mutually taking the roles of programs and references.
Chapter 20

Conclusions

— A conclusion is the place where you got tired thinking.

Martin Henry Fischer

Two types of automatic testing tools have been explored. The first type, called specification-based testing tool, uses specifications as an oracle to determine the outcome of the test.

For this tool, the effects of writing the specification in a traditional programming language has been investigated. In particular, the effects of using program-based coverage criteria (branch coverage) for specifications have been studied.

In general, it was found that it is not possible to achieve full coverage using a program-based criteria on specifications. This is because program-based specifications are likely to contain so-called false subdomains. Such subdomains express incorrect program behavior. Consequently, correct programs should not cover these subdomains.

Since it is difficult to estimate how much of the specification is made out of false subdomains, it is also difficult to use only the specification coverage as a stopping criteria.

Experiments using a prototype implementation of the tool have been conducted. One important fact that was revealed here is that the tool is incapable of detecting situations where the specification contains functionality that is not implemented by the program. Mathematically speaking, the tool cannot detect that a specification $s$ contains a program $p$, that is, $s \supset p$.

The outcome of this is low coverage, because the parts, that is, subdomains, of the specification that are not implemented by the program are not covered. Since the specification most likely contains false subdomains, to our knowledge it is not possible to determine whether the low degree of coverage is due to false subdomains or that $s \supset p$. 

171
Chapter 20 Conclusions

This flaw can easily be fixed by a slightly different design of the tool referred to as a parallel design. The initial version of the tool, which is suitable for random testing, uses a serial design. Serial and parallel describe whether the specification and program needs to be executed in a serial or parallel manner.

Using a parallel design of the tool fixes the problem of not being able to detect $s \supset p$. However, using parallel testing tools with random input generators comes with a performance penalty. If systematic generators are used there is no penalty.

The fault-finding effectiveness of a prototype tool using various input generation strategies has been studied. Random input generation has been compared with both hand- and machine-generated input sets.

The most notable result was that even a very simple implementation of the random input generator finds a high percentage of the faults, roughly 95%. Tuning the random generator just a bit more results in finding 100% of the faults.

Furthermore, the fact that program-based coverage yields false subdomains for specifications is shown not to be that dramatic. By plotting the program coverage and specification coverage as a function of the number of inputs it can be seen that the two follow each other very well, with the program coverage being asymptotically higher.

By looking at the increase in the program’s and the specification’s rates of coverage, and the degree of program coverage one can decide when to stop testing or if the input generator needs to be improved.

The second tool studied in this part is a program-based testing tool. Here a reference program is used as an oracle. The reference program can be a prototype or another version of the program under test.

Program-based testing tools do not have either of the two drawbacks of the specification-based tool. Since the reference program is a plain program, false subdomains are not an issue. Furthermore, program-based testing tools tool are inherently parallel in their implementation, thus both $r \supset p$ and $r \subset p$ can be detected, where $r$ is a reference and $p$ is a program.

A prototype of the program-based tool using random input generation has also been evaluated. As with the specification-based tool, the degree and rate of program (and reference) coverage together with the number of generated inputs give a good indication whether to stop testing or improve input generation.

The fault-detection effectiveness of random testing is not affected by the adequacy criterion in use in the same way as with systematic testing. In systematic testing the criterion selects which tests to run from all
tests. In random testing plenty of tests are actually run. The criterion is consulted to know which tests to save and to give feed back to the tester on how strong the input generator is.
References


References


References


References


References


References


No 461 Lena Strömbäck: User-Defined Constructions in


No 582 Vanja Josifovski: Design, Implementation and
<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Title</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>589</td>
<td>Rita Kovordány</td>
<td>Modelling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity</td>
<td>91-7219-506-1</td>
</tr>
<tr>
<td>592</td>
<td>Mikael Ericsson</td>
<td>Supporting the Use of Design Knowledge - An Assessment of Commenting Agents</td>
<td>91-7219-532-0</td>
</tr>
<tr>
<td>593</td>
<td>Lars Karlsson</td>
<td>Actions, Interactions and Narratives</td>
<td>91-7219-534-7</td>
</tr>
<tr>
<td>595</td>
<td>Jörgen Hansson</td>
<td>Value-Driven Multi-Class Overflow Management in Real-Time Database Systems</td>
<td>91-7219-542-8</td>
</tr>
<tr>
<td>596</td>
<td>Niklas Hallberg</td>
<td>Incorporating User Values in the Design of Information Systems and Services in the Public Sector - A Methods Approach</td>
<td>91-7219-543-6</td>
</tr>
<tr>
<td>597</td>
<td>Vivian Vimarlund</td>
<td>An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories</td>
<td>91-7219-544-4</td>
</tr>
<tr>
<td>598</td>
<td>Johan Jenvald</td>
<td>Methods and Tools in Computer-Supported Taskforce Training</td>
<td>91-7219-547-9</td>
</tr>
<tr>
<td>607</td>
<td>Magnus Merkel</td>
<td>Understanding and enhancing translation by parallel text processing</td>
<td>91-7219-614-9</td>
</tr>
<tr>
<td>611</td>
<td>Silvia Coradeschi</td>
<td>Anchoring symbols to sensory data</td>
<td>91-7219-623-8</td>
</tr>
<tr>
<td>618</td>
<td>Jimmy Tjäder</td>
<td>Systemimplementering i praktiken - En studie av logiker i fyra projekt</td>
<td>91-7219-657-2</td>
</tr>
<tr>
<td>637</td>
<td>Esa Falkenroth</td>
<td>Database Technology for Control and Simulation</td>
<td>91-7219-766-8</td>
</tr>
<tr>
<td>660</td>
<td>Erik Larsson</td>
<td>An Integrated System-Level Design for Testability Methodology</td>
<td>91-7219-890-7</td>
</tr>
<tr>
<td>688</td>
<td>Marcus Bjäreland</td>
<td>Model-based Execution Monitoring</td>
<td>91-7373-016-5</td>
</tr>
<tr>
<td>689</td>
<td>Joakim Gustafsson</td>
<td>Extending Temporal Action Logic</td>
<td>91-7373-017-3</td>
</tr>
</tbody>
</table>


Linköping Studies in Information Science


