View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CiteSeerX

Master Thesis

Felix Kurth

Automated Generation of Unit Tests from UML Activity Diagrams using the AMPL Interface for Constraint Solvers

January 3, 2014

supervised by:
Prof. Dr. Sibylle Schupp
Prof. Dr. Ralf God

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

Statutory Declaration

I, Felix Kurth, declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. Neither this thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, January 1, 2014

....
Felix Kurth

Abstract

Testing is one important way of validating and verifying the behaviour of software artefacts. With the use of Model–Based Engineering comes the need to ensure that the implementation derived from models actually behaves like specified in the corresponding models.
In this thesis, we generate unit tests from UML activity diagrams modelling C components. We use symbolic execution to transform embedded OCL constraints along a control flow path into a rigorous mathematical program expressed in '**A** **M**athematical **P**rogramming **L**anguage' (AMPL). We derive test data from the mathematical program by having it solved with state–of–the–art constraint solvers. The generated tests will satisfy control flow–based coverage criteria on the used models. Since unit tests using test data at the bounds of path constraints do have a higher probability to detect faults, we also use boundary value analysis for testing. A special focus is on allowing mixed integer non–linear programming as well as logical formulas in OCL constraints because we want to put as little restrictions as possible on the used test models.

Contents

1 Introduction **1**
1.1 A Real World Problem at Airbus . . . 1
1.2 Outline of this Thesis . . . 2
1.2.1 Model Transformations . . . 2
1.2.2 Symbolic Execution . . . 3
1.2.3 State–Of–The–Art Constraint Solver Implementations . . . 3
1.2.4 Results . . . 3
1.3 Organization . . . 3

2 Preliminaries **5**
2.1 Literature Review . . . 5
2.1.1 Test Models . . . 5
2.1.2 Test Generation . . . 6
2.1.3 Test Data Generation . . . 7
2.2 Mathematical Foundations . . . 8
2.2.1 Characteristics of Problems and Solver Methods . . . 9
2.2.2 Constraint Satisfaction Problem . . . 10
2.2.3 Optimisation Problem . . . 11
2.2.4 Convex Optimisation Problem . . . 12
2.2.5 Linear Programming . . . 12
2.2.6 Boolean Satisfiability Problems . . . 13
2.2.7 Satisfiability Modulo Theories . . . 14
2.2.8 Properties of the Search Space . . . 14
2.3 The AMPL Modelling System . . . 16
2.3.1 Command Language . . . 16
2.3.2 Modelling Language . . . 16
2.3.3 Entering Data . . . 18

3 Generating Unit Tests from a UML Activity **19**
3.1 General Overview of the Workflow . . . 19
3.2 Normalisation . . . 21
3.2.1 Design Rules for the Test Model . . . 21
3.2.2 A Meta Model Suitable for Automated Unit Test Generation . . . 22
3.2.3 Transforming an UML Activity Diagram to an Activity Test Case Graph . . . 25
3.2.4 Adding Continuity Constraints . . . 28
3.3 Rigorous Mathematical Programming . . . 29
3.3.1 Introductory Example . . . 29
3.3.2 Transforming a TCGActivity to an AMPL Model . . . 30
3.3.3 Transforming TCGVariables . . . 30
3.3.4 Transforming LocalPostConditions . . . 32
3.3.5 Transforming Guards . . . 33

3.3.6 Specifying Control Flow Paths in the AMPL Data . . . 33
3.4 Abstract Test Case Generation . . . 34
3.4.1 Path Tree . . . 35
3.4.2 Path Search . . . 35
3.4.3 Early Infeasible Path Elimination . . . 37
3.5 Specific Test Data Generation . . . 38
3.5.1 The Unit Test Meta Model . . . 39
3.5.2 Solving a Constraint Satisfaction Problem with AMPL . . . 39
3.5.3 Storing the Variable Values in a Unit Test Case Model . . . 40
3.5.4 Review of Available Solvers . . . 41
3.5.5 Warm Start Capabilities . . . 41
3.5.6 Boundary Value Analysis . . . 43
3.6 Unit Test Synthesis . . . 43

4 Evaluation 47
4.1 Experiment Description . . . 47
4.1.1 Runtime Measurement . . . 47
4.1.2 Mutation Testing . . . 49
4.2 Academic Examples . . . 49
4.2.1 Triangle Classificator (LP) . . . 49
4.2.2 Triangle Classificator with Logical Constraints (SMT) . . . 51
4.2.3 Binary Counter (SAT) . . . 52
4.2.4 Exploding Tyres (MINLP) . . . 54
4.3 Case Study PAX Model (MILP) . . . 56
4.3.1 Manual Adaptation . . . 57
4.3.2 Runtime Measurement Results . . . 57
4.4 Verification of the Implementation . . . 61
4.5 Limitations . . . 61
4.5.1 Theoretical Limitations . . . 62
4.5.2 Limitations of the Implementation . . . 62
4.5.3 Limitations of Tool Support . . . 63

5 Outlook and Summary 65
5.1 Outlook . . . 65
5.2 Summary . . . 66

1 Introduction

Model–Based Engineering is a recent technology in the domain of electrical and software engineering. In Model–Based Engineering, the specification of a system is developed as a model. Graphical modelling languages can give a quick and intuitive overview of a system. At the same time, one can model very detailed and describe a system bit by bit with formal modelling languages. Nevertheless, creating a model of a system in practice is an effort that needs to pay off. One way to get a benefit from Model-Based Engineering is to use the design model to automate other tasks in software development. One important task is testing. In this thesis, we will develop an algorithm to automatically generate relevant test cases and even working unit test code from the model. We assume a C function as the implementation to be tested. We will focus on UML activity diagrams modelling the control flow and behaviour of the C function in a component–based architecture as test model. Since it is not a trivial task to transform a UML activity diagram into C/C++ unit test code, we will split the task up into subtasks, specify the interfaces between the different steps, and give algorithms for each step. Due to its generic architecture, our algorithm can be adapted for different input models, output languages, coverage criteria, and constraint specification languages by modification of only one or two of the subtasks.

1.1 A Real World Problem at Airbus

At Airbus Buxtehude, Model–Based Engineering shall soon replace the traditional Requirement–Driven Engineering approach. There are some benefits that we hope to be able to realize with Model-Based Engineering. One benefit is the automatic generation of source code from the design model. Another way to profit from Model–Based Engineering is to use the specification as a test model from which unit test code for the software implementation could be derived. In Figure 1.1, we visualise the idea of using one activity diagram to generate both implementation code and unit test code from it. There is a simple activity diagram in the upper part of the figure, and we show the function implementation that could have been generated from the activity diagram below on the left hand side. On the right hand side, the generated test suite is shown containing two tests. Each test is testing one control flow path.
UML activity diagrams will be used to model the behaviour and the control flow of C functions. The company Atego® has already built a proprietary code generator for Airbus Buxtehude, which generates a folder structure, C source code files, and C header files from a UML model, and fills function bodies with code generated from UML activities. The fact that the code is generated, however, does not make it fault–free. We still need to validate the generated code by unit tests. The task in this thesis is to demonstrate how engineers can be supported in building the unit tests for the generated code. We decided to automate the generation of C++ unit tests from UML activity diagrams.
It is common knowledge that one should not generate both the source code and the unit test from the same model. The design model needs to be independent from the test model. Otherwise, one ends up testing the implementation against itself, while errors in the model that have spread to both the source code and the unit test are overlooked.

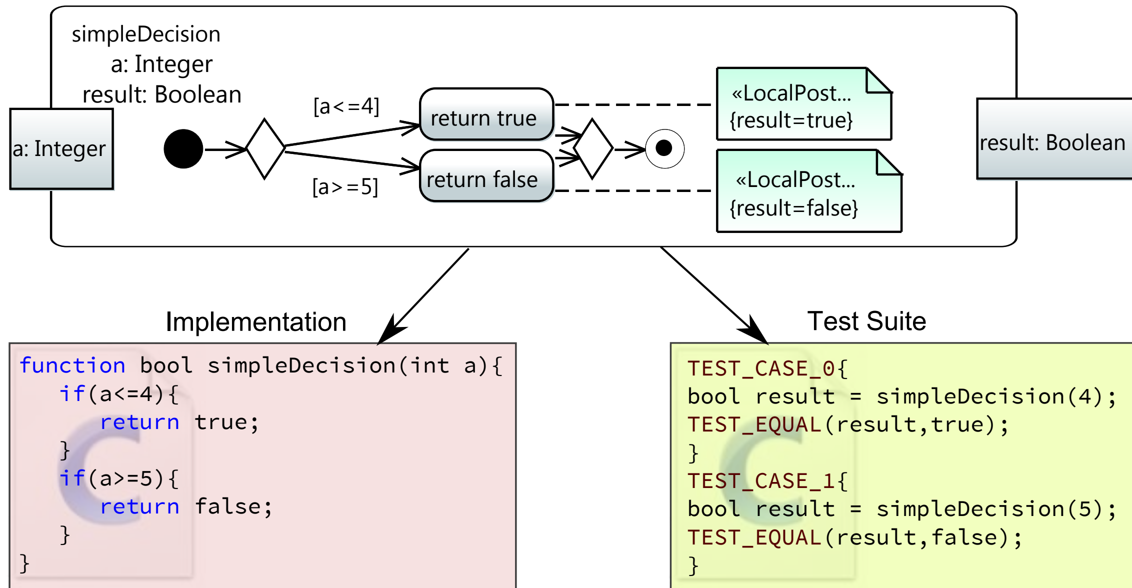

Figure 1.1: Example of an activity diagram with a corresponding C implementation and C unit tests

On the other hand, building a test model completely independent from the design model imposes much additional effort. The decision was to share the structure of the activity diagram between code generation and test generation. While the code generation uses procedural C code snippets that are embedded in the model, the test generation only relies on embedded declarative OCL constraints. Since we are using two different programming paradigms as input for code generation and test data generation, we assume the generated C implementation to be independent from the generated unit tests.

1.2 Outline of this Thesis

This thesis focuses on generating unit tests from UML activity diagrams with embedded OCL constraints. Our approach is based on graph search algorithms to generate abstract test scenarios. We use symbolic execution and constraint solving to make the test model executable and automatically find useful test data to test the implementation. We put some emphasis on mixed–integer–non–linear arithmetic constraints and will evaluate the limits of state–of–the–art solver implementations.

1.2.1 Model Transformations

We transform a UML activity diagram into a rigorous, executable, mathematical representation. We decided to use multiple subsequent model transformations to carry out this transformation stepwise. In Section 3.2 we will specify how the activity diagram is transformed to an intermediate representation. During this transformation we can check design rules and ensure properties of the intermediate representation that are necessary in subsequent steps. From this intermediate representation we describe in Section 3.3 a model–to–text transformation to '**A** **M**athematical **P**rogramming **L**anguage' (AMPL).

1.2.2 Symbolic Execution

In order to make UML activity diagrams executable for test data generation we will transform all relevant constraints embedded in the UML model into a mathematical program. Executing a certain control flow path in the activity diagram means generating the corresponding mathematical program. The mathematical program will be stated in AMPL syntax. A mathematical program in AMPL consists of an AMPL model and AMPL data. One activity diagram will be transformed into one AMPL model and the currently executed control flow path can be specified in the AMPL data.
When building the AMPL model from the activity diagram, we have to be careful that the semantics of the resulting AMPL model corresponds with the semantics the modeller originally had in mind. We support the user by automatically adding constraints that would be cumbersome to make explicit manually.

1.2.3 State–Of–The–Art Constraint Solver Implementations

A great advantage of a commonly used mathematical programming language is that industrial–strength solvers are available. The AMPL program corresponding to a possible control flow path in the activity diagram will be solved with a state–of–the–art solver. Then we obtain test data from the results of the constraint solver. The generated test data includes possible input arguments for the C function that implements the activity diagram. Also, all expected return values as well as values for class properties before and after the execution of the activity diagram are included.
A suitable solver needs to be selected depending on the problem formulation. Not every solver can solve every AMPL program, and not every AMPL program is an instance of an easy–to–solve problem. An overview of the problems that are considered throughout this thesis can be found in Section 2.2.

1.2.4 Results

A practical result of this thesis is an Eclipse plug-in that automatically generates unit tests from activity diagrams. The plug-in comes with a set of example models. Each example is an instance of a different mathematical problem and thus needs an appropriate solver to be solved successfully. Moreover, we performed an industrial case study using a model of the PAX–Call system, which is part of an Airbus product. We added the missing OCL constraints and successfully generated test cases for the largest activity diagram within the model. Finally, we experimentally determined the scalability of the implementation.

1.3 Organization

The rest of this thesis is organized in three chapters. In the chapter Preliminaries, we will refer to related work (Section 2.1), give an overview of the theory of mathematical programming and constraint programming (Section 2.2), and explain the mathematical programming language AMPL.
In the chapter Generating Unit Tests from a UML Activity, we first give a short overview of the overall algorithm we are proposing (Section 3.1), and then explain the five subsequent steps of the algorithm (Sections 3.2 – 3.6). Each of them can be viewed as model–to–model, model–to–text, or text–to–model transformation.
The chapter Evaluation presents the example collection (Section 4.2) that has been tested with the implemented Eclipse plug-in and elaborates on the strengths of different solvers.

We measured the runtime of our algorithm with different solvers and performed mutation tests with the generated test suits. The case study (Section 4.3) can be found in this fourth chapter as well. We experimentally determined the scalability of our implementation within our case study. Finally, we will also sum up problems that could not be solved within this thesis (Section 4.5).

2 Preliminaries

Throughout this thesis we will develop a complicated algorithm. We will rely on existing technology and knowledge throughout the third and the fourth chapter. Thus, in this chapter we present a minimal mindset that is helpful to understand the presented algorithm and reproduce our argumentation.
First, we will present in Literature Review what others did to solve problems similar to ours. Then we will provide a little background knowledge on the computability of different specializations of the constraint satisfaction problem and the algorithms that are applied to the considered problems in Mathematical Foundations. '**A** **M**athematical **P**rogramming **L**anguage' (AMPL) is used in the presented algorithm; thus, we will explain the basic concepts of AMPL in The AMPL Modelling System.

2.1 Literature Review

Model–Based Testing emerged in the 1990s. A variety of different modelling paradigms and languages as well as test generation methods have been proposed since then. Model–Based Testing has also been applied in different scopes, such as, for unit testing or system testing. Mark Utting offers in [1] a possible classification of approaches to Model–Based Testing in seven orthogonal dimensions. In Figure 2.1, we see an overview of possible categorisations of Model–Based Testing approaches. Any work on Model–Based Testing roughly consists of three important steps: Specifying what the test model is, generating some abstract representation of test cases, and refining the abstract test cases into concrete executable test cases by determining the necessary input data and potentially also providing an oracle value for the expected output. The rest of this section is structured according to those three steps and in each subsection we will shortly present the propositions made in different publications and use Mark Utting's taxonomy to classify them and to position our own thesis relative to them.

2.1.1 Test Models

In [2] Lackner and Schlingloff propose to build a test model independently from the implementation. Their test models modelled the behaviour of the system. They present three transition–based modelling paradigms suitable for automated test case generation: Abstract State Machines, UML2 State Machines, and Timed Automata. Early research on Model–Based Testing performed by A. Abdurazik and Jeff Offutt [3] also proposed using UML state charts as test model to generate system level test cases from them. The same authors proposed in [4] UML collaboration diagrams as test models. Stephan Weißleder and Dehla Sokenou propose in [5] using UML state models together with UML class models, and OCL pre– and post–conditions as test models.
It is very common to use state machines as the test model, while less research is performed on generating unit tests from UML activity diagrams. Wang Linzhang et al. propose in [6] using UML activity diagrams. They propose to generate test cases from an Activity modelling an operation in the design model. They also claim to have implemented a

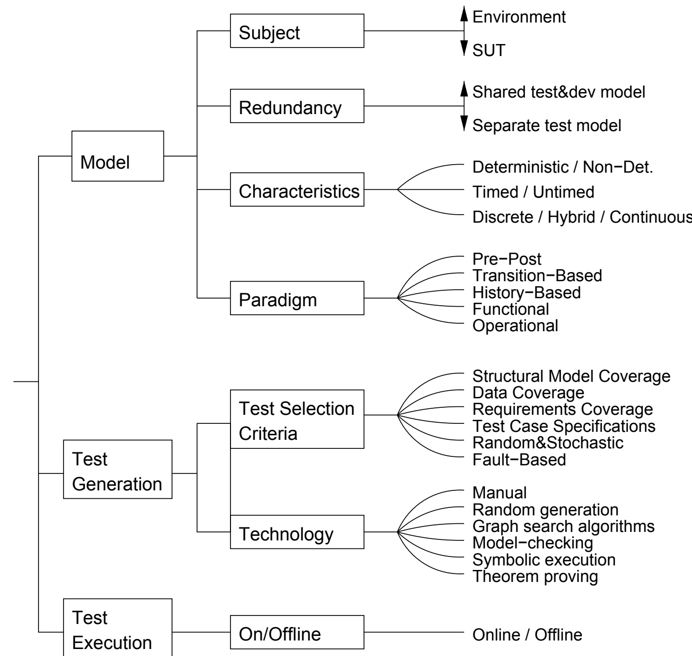

Figure 2.1: Overview of the taxonomy of Model-Based Testing approaches [1]

proof–of–concept tool called UMLTGF, which, unfortunately, was unavailable for review. In [7] Debasish Kundu and Debasis Samanta consider UML activity diagrams specifying use cases of the system under test.
Achim D. Bruckner and Burkhard Wolff propose in [8] a full theorem prover and constraint solver based approach to test generation. Thus, their input models are entered by means of a functional programming paradigm. They consider models of the system under test as well as additional information about how the system shall be tested as test model.
For our own work we use as test model some details from the UML class model and an UML activity diagram modelling the inner workings of the unit under test. The unit under test is in our case a C function. The model we use for test generation shares its control flow structure with the model used to generate the implementation similar to [6]. The information needed to be able to derive test data is added separately to the test model, not shared with the design model, in the form of OCL pre– and post–conditions and guards. The use of OCL constraints is similar to the work of Stephan Weißleder and Dehla Sokenou [5]. Furthermore, our input models are deterministic, untimed, and discrete.

2.1.2 Test Generation

In test generation, coverage criteria are a widely accepted means to measure the quality of a test suite and steer the test generation. Aynur Abdurazik and Jeff Offutt proposed in [3] a handfull of coverage criteria concerned with the structure of the test model. By structure we mean the nodes and the arcs of the state chart they used as input. They present path search algorithms to find transitions sequences fulfilling the proposed coverage criteria. Puneet E. Patel and Nitin N. Patil compare in [9] two different path finding algorithm selecting control flow paths within an activity diagram as abstract test cases. Similarly, [7] and [6] propose a kind of graph search as technology to find suitable test cases.
Stephan Weißleder and Dehla Sokenou presented in [5] a data oriented approach to select

test cases. They propose to use abstract interpretation to derive partitions of the domain of input values. They then propose to select values at the boundaries of the computed partitions as test data. In Stephan Weißleders Ph.D Thesis [10] we found the most detailed and complete collection of different model–structure and data–oriented coverage criteria. Furthermore, a formal definition of the coverage criteria is given. The source code of ParTeG, the proof–of–concept tool associated with this thesis, is freely available and we were able to test it. It uses graph search in combination with abstract interpretation and a comprehensive framework allowing the user to steer which coverage criteria the generated test cases will adhere to.
A comprehensive approach to automated test generation based on automated theorem proving and constraint solving is presented by Achim D. Brucker and Burkhart Wolff in [8]. They generate test cases by applying natural deduction to a test case specification given in the input language of a theorem prover. The so called test theorem resulting from the natural deduction is a sequence of executable statements that serves as abstract test case. With constraint solving concrete test data is computed for each abstract test case.
In our thesis we use an easy to implement path search algorithm to generate abstract test cases. The path search is supported by symbolic execution and constraint solving to determine infeasible paths.

2.1.3 Test Data Generation

Chen Minsong et al. [11] combine random test case and data generation with graph structure oriented coverage criteria. Their approach is to generate random executable unit tests for a Java implementation and matching the execution traces of the randomly generated unit tests with the control flow paths found in an activity diagram. They then select a subset from the test cases, which covers all simple paths in the test model. A simple path is a control flow path without cycles. This is a randomised approach to test data generation combined with a structural model coverage criterion. Jan Peleska et al. [12] describe a combination of abstract interpretation with constraint solving using an SMT solver in order to generate test data. They consider the execution of a test model as consecutive execution of a state transition function, which changes the state of the system. They have the iterated state transition function and further predicates solved by an SMT solver and support the solver by providing it with bounds for some variables deduced via abstract interpretation.
Using OCL as input and trying to obtain test data by solving OCL constraints there have been several proposals on how exactly to find models making an OCL formula true. Shaukat Ali et al. proposed in [13] using evolutionary or genetic algorithms to search for potential solutions of OCL constraints. Matthias P. Krieger and Alexander Knapp proposed in [14] using a SAT solver to find instantiations of variables fulfilling OCL constraints. They suppose to transform OCL formulas into boolean formulas with bounded quantifiers and uninterpreted functions. Consequently they use a model finder, based on a SAT solver, capable of handling those formulas to find assignments to the variables.
In [15] Jan Malburg and Gordon Fraser propose a hybrid approach. On the top level they use a genetic algorithm evolving a population of candidate test data internally they provide guidance to the genetic algorithm by providing a special mutation operator performing dynamic symbolic execution. Also the white–box unit test tool PEX [16] from Microsoft® research is based on dynamic symbolic execution. Both publications, [15] and [16], propose to execute the implementation with random input values and generate new input values by collecting all path conditions along the executed control flow. Then they negate one of the path conditions and use a constraint solver to find a solution for the new constraint

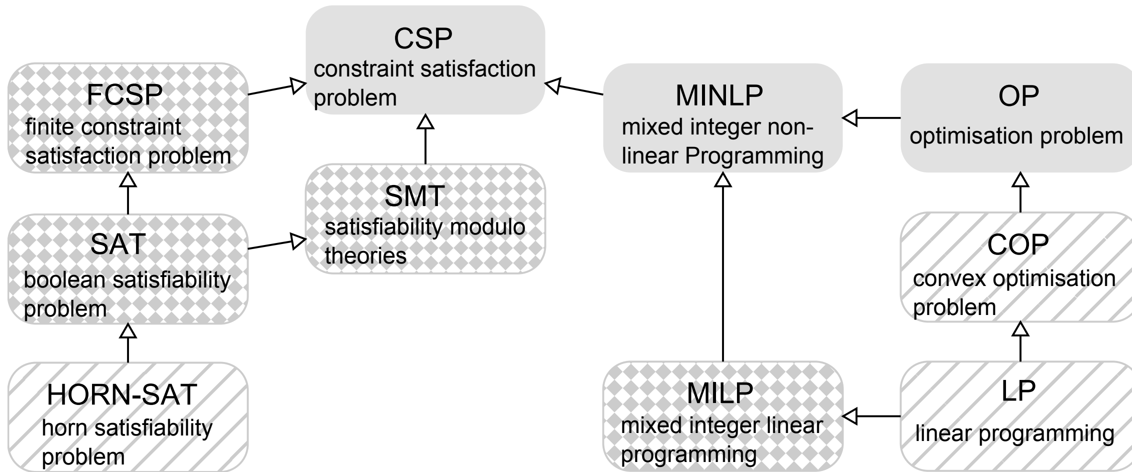

Figure 2.2: Generalisation and specialization dependencies between different problems. Decidable problems for which tractable algorithms exist have a lined background; decidable problems that are NP–complete have a squared background; undecidable problems have a grey background.

system. The generated solution is new input data which is guaranteed to take another control flow path in the program.
Our own test data generation approach is based on symbolic execution of control flow paths in an activity diagram and constraint solving using a collection of solvers through a common interface. We collect all the constraints along a control flow path and represent them in a constraint satisfaction problem which is solved by a constraint solver.

2.2 Mathematical Foundations

Constraint solving is a common technique to derive answers to particular questions. Depending on the constraint satisfaction problem instance there might exist an algorithm and even a software implementation that can compute the answer to our question. A major task in this thesis is to transform a control flow path in an activity diagram into a mathematical program, that is, a constraint satisfaction problem, and to apply a constraint solver.
Depending on the used test model the resulting mathematical program will be an instance of one of the problems presented in this section. Since currently existing solvers are always specialized on particular problems one needs to understand of which problem this mathematical program is an instance. Certain problems are a specialization of more general problems. In Figure 2.2, we give an overview of the problems presented in this Section and visualise which problems are generalisations or specializations of each other. An arrow from A to B means that B is a generalisation of A.
Some mathematical programs are harder to solve than others. Instances of more general problems are usually harder to solve than instances of a specialized problem. More specialized problems restrict the expressiveness of constraints that are allowed in its instances. We are especially interested in problems that allow our test models to be most expressive and are just specialized enough that we can find a useful solver for the resulting mathematical program. Those are the user–friendly problems. When we do not need the expressiveness of a general problem we can use specialized solvers computing solutions of

the mathematical programs very quick. Those are the easy–to–solve problems.
In Characteristics of Problems and Solver Methods we introduce definitions that will be used to characterise problems and methods and explain the implications of those characteristics for constraint solving. For every presented problem we will mention their characteristics. In Sections 2.2.2–2.2.7 we present different means to express constraints and introduce a variety of problems. In Properties of the Search Space we will present properties of variables and introduce further problems.

2.2.1 Characteristics of Problems and Solver Methods

We characterise the solvers used by two orthogonal properties: the decidability of the problem they solve, and the tractability of the method they implement.

Decidability of Problems

Not every problem presented in the following subsections will be decidable. We denoted undecidable problems with a solid grey background in Figure 2.2.

Definition 1 (decidability) *A problem is decidable if an algorithm solving that problem exists.*

When the problem is decidable we can assume that there is an exact solver available. An exact solver will tell for a problem instance that the problem has no solution only if there really is no solution. If it returns a solution we assume it is correct.
When a problem is undecidable, on the other hand, that means there can never be an algorithm solving all instances of this problem. Still, there exist heuristics that try to handle undecidable problems. A heuristic may find a solution for a given problem instance if it is lucky, or just terminate without finding a solution although one exists, or it might run forever, or even just return a wrong answer. For instances of undecidable problems a heuristic telling that there is no solution just means that this particular search was not lucky and returned answers might not be correct solutions. For example, it can return a sub–optimal point for an optimisation. In most cases the answer of a heuristic is good enough to be useful in practice. If a heuristic did not find a solution one could start again with different parameters for the heuristic, for example, another starting point, or one could use another solver. In practice one will run a heuristic until a certain time limit has been reached.

Tractability of Algorithms and Heuristics

Algorithms and heuristics can be tractable or intractable.

Definition 2 (tractability) *A method is tractable if its worst case runtime is in $\mathcal{O}(u^n)$ for a constant n.*

For some problems like the *linear programming* (LP) explained in Section 2.2.5 there exist tractable algorithms. In Figure 2.2, problems for which a tractable algorithm exists have a lined background. For other problems there can be instances that are NP–complete, for example, the *boolean satisfiability problem* (SAT) presented in Section 2.2.6. Any algorithm solving an NP–complete problem will be intractable. Problems for which no tractable algorithm exists are marked with a squared background in Figure 2.2.
If the algorithm or heuristic used to solve a certain problem instance is tractable, we know

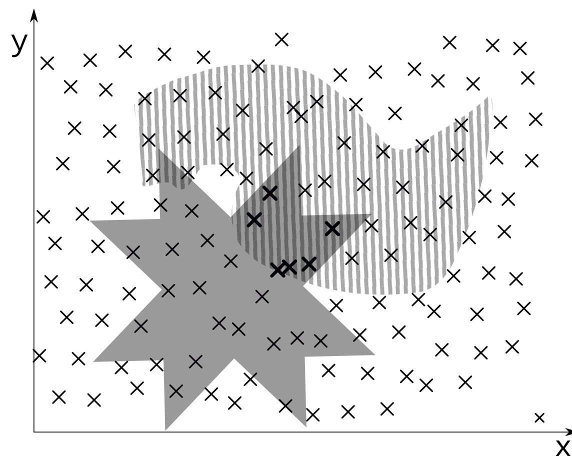

Figure 2.3: Two–dimensional search space and three different subsets forming a constraint satisfaction problem

that we can expect an answer even for large problem instances after a moderate amount of time. For intractable algorithms the runtime might grow exponentially and therefore make it practically infeasible to compute an answer to a given problem instance. For practical problems it is often the case that an intractable algorithm or heuristic has a non–exponential average runtime. The simplex algorithm is a good example for such an algorithm. Its worst case runtime is exponential but for problems occurring in practice its runtime grows linear with the number of constraints.

Further Considerations

Among the decidable problems those with a tractable algorithm are especially easy–to–solve. Whenever it is possible to state a certain problem as an instance of such an easy–to–solve problem it is advisable to do so. If all constraints in the used test model are instances of an easy–to–solve problem, one can be sure that there is a suitable solver solving the problem correct in a moderate amount of time with respect to the size of the problem.
Restricting the expressiveness of constraints to decidable problems can be a strong limitation for the user, and is not strictly required for constraint solving. It is not always easy or impossible to express constraints as a problem instance of a decidable problem that can be solved by a tractable algorithm. Consequently we will also consider using solvers for decidable problems with intractable algorithms as well as solvers for undecidable problems implementing tractable and intractable heuristics. A solver is useful in practice if it has a fair chance to produce a useful answer and if it reports useful answers for practical instances in average after a moderate amount of time.

2.2.2 Constraint Satisfaction Problem

In a *constraint satisfaction problem* (CSP) the task is to assign values $v(x_i)$ to a set of variables $X = (x_1, \ldots, x_n)$ from a *search space* $S = D_1 \times \ldots \times D_n$ such that all relations $c_1, \ldots, c_k$ between the variables hold. We call those relations $c_1, \ldots, c_k$ *constraints*. D_i is called the *domain* of x_i, n is the number of variables, and k the number of constraints. We

can formalise a constraint satisfaction problem as shown in equations (2.1)–(2.3) [17].

$$c_i \subseteq S \qquad \forall i \in [1 \dots k] \tag{2.1}$$
$$\text{find:} \quad v(x_i) \in D_i \qquad \forall i \in [1 \dots n] \mid \tag{2.2}$$
$$\text{subject to:} \quad (v(x_1), \dots, v(x_n)) \in c_j \qquad \forall j \in [1 \dots k] \tag{2.3}$$

Definition 3 (feasible solution) *Any point in S fulfilling all constraints is called a feasible solution.*

Definition 4 (feasible set) *The set of all feasible solutions is called the feasible set.*

The feasible set is the intersection of all constraints. If the feasible set is the empty set we call the problem infeasible. Solving a constraint satisfaction problem means finding one feasible solution.
Practically, constraints are often specified as boolean terms over a subset of X. For example $x_1 = x_2$ or $0 \leq x_1 \leq 1$. It is possible to extend any relation between x_1 and x_2 to a n-ary relation between $x_1, \dots, x_n$. In the extended relation all possible values are permitted for the variables not contained in the term.
In general, there are no restrictions on the domains of the variables. They could be continuous, discrete, finite, and infinite. Well–known domains are integer, real, boolean but city names or all triangles similar to an equilateral triangle could also form a domain. Similarly, the constraints can have arbitrary properties. Even a fractal set would be acceptable as constraint.
In Figure 2.3, we can see a two–dimensional continuous search space and three sets. One that could be assembled by a union of intersections of half–spaces, another that has an arbitrary shape, and the last set that consists of discrete points marked by the crosses. Any of the points within the intersection of those three sets is a solution to the corresponding CSP.
The CSP in its general form is undecidable but there are decidable specializations with a few restrictions on the kind of constraints and domains. For example a CSP with all variables in $\mathbb{B}$ is decidable.

2.2.3 Optimisation Problem

An *optimisation problem* (OP) is a constraint satisfaction problem with a real–valued *objective function* $f : S \mapsto \mathbb{R}$ where constraints are specified in terms of real–valued functions $g_1, \dots, g_k : S \mapsto \mathbb{R}$. We call those functions *constraint functions*. The search space is $\mathbb{R}^n$. The task is to find a feasible point that minimises the objective function f.

$$\text{minimise:} \quad f(X) \tag{2.4}$$
$$\text{subject to:} \quad g_i(X) \leq 0 \qquad \forall i \in [1, \dots, k] \tag{2.5}$$

We distinguish two kinds of optima, the local optimum and the global optimum. Global optima are in general much more difficult to find than local optima.

Definition 5 (local optimum) *A local optimum is a feasible solution that minimises the objective function within a neighbouring set of feasible solutions.*

Definition 6 (global optimum) *A global optimum is a feasible solution that gives the minimal value for the objective function among all feasible solutions.*

In practice equations are also used as constraints. In equation (2.5), $\geq$ is permitted as well as an arbitrary constant on the right–hand side.
In this thesis only linear objective functions are considered. For linear objective functions that are not constant any optimum can be found on the boundaries of the feasible set. That is why the word *boundary value* will be used as an equivalent for local optimum. Using a constant objective function relaxes the problem, since with a constant objective function every feasible point is a global optimum.
The OP is undecidable, but there are very successful heuristics available. One way to tackle an OP is to add quadratic penalty terms to the objective function, which are growing when a constraint is violated. Then one selects an arbitrary starting point and applies a descend method such as Newton's method to find a local minimum of the augmented objective function. One can try this local search with a descend method multiple times from different starting points and then select among the points found the one that gives the lowest value for the objective function. With a high likelihood, this algorithm will find a feasible point, and the global optimum could have been among the computed local optima.

2.2.4 Convex Optimisation Problem

The *convex optimisation problem* (COP) is a decidable specialization of the optimisation problem for which tractable algorithms exist [18].

Definition 7 (convex set) *A set C in a vector space S is said to be convex iff for all points x and y in C and all $t \in [0,1]$, the point $(1-t)x + ty$ is in C.*

Definition 8 (convex function) *A function f is convex iff for any two values x and y the inequality $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$ with $\theta \in [0,1]$ holds.*

A convex optimisation problem is an optimisation problem with a convex objective function and convex constraint functions. The intersection of convex sets is a convex set itself and the set $\{x \in \mathbb{R}^n | g_i(x) \leq 0\}$ is a convex set if g_i is a convex function. Consequently the feasible set is also convex. For convex optimisation problems every local optimum is a global optimum.
The convex optimisation problem is decidable and additionally there are tractable interior point methods solving instances of that problem. Interior point methods first need a feasible point as starting point. The objective function is augmented by logarithmic barrier functions for each constraint with a form factor to steer the steepness of these barriers. Now a descend method is applied to find the lowest point. Then repeatedly the barriers are made steeper and the last point found is used as a new starting point. The repetition ends when a stop criterion is fulfilled [18].

2.2.5 Linear Programming

Within the class of convex optimisation problems there are many special cases. One specialization of the convex optimisation problem is *linear programming* (LP). In linear programming every constraint can be represented as linear inequality and the objective function is an affine function. The canonical form of a linear program consists of a matrix $\mathbf{A}$ and two vectors c and b as shown in the equations (2.6)-(2.7). The relation $\leq$ is evaluated componentwise.

$$\text{minimise:} \quad c^T X^T \tag{2.6}$$
$$\text{subject to:} \quad AX^T \leq b \tag{2.7}$$

By removing the objective function or setting $c = 0$ the problem is slightly relaxed. Formulations containing also equations in the constraints on some components of X can be transformed to an equivalent problem in the canonical form. Also, maximisation of the objective function might be required instead of a minimisation. Further the relation $\geq$ might replace the $\leq$ for some components of the constraint. In practice we will use all of these formulations.
An example of a linear program is given in equations (2.8)-(2.9).

$$\begin{pmatrix} 1 & -2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} \tag{2.8}$$

$$\begin{pmatrix} 0 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \vec{0} \tag{2.9}$$

Linear programs can efficiently be solved with an implementation of the simplex algorithm [19]. Although its worst case runtime is exponential the average runtime grows linearly with the number of constraints. For a geometric interpretation of the simplex algorithm we interpret the constraints as a set of hyper–planes in an n–dimensional space. The feasible region is a polyeder whose faces are within the hyper–planes defined by the constraints. The simplex algorithm starts at an arbitrary exposed point of this polyeder and then moves along one edge to another exposed point with a better objective value. This step is repeated until the exposed point with the minimal or maximal objective value is found.

2.2.6 Boolean Satisfiability Problems

In a classic *boolean satisfiability problem* (SAT) all variables are within the boolean domain and can take the values $true$ and $false$ or 0 and 1 respectively. The constraint relations are expressed by a boolean formula. The task is to find an assignment to the variables that makes the formula true. A simple example of a boolean formula is given in equation (2.10).

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2 \tag{2.10}$$

Any boolean formula can be normalised to a conjunction of disjunctive clauses, the so–called *conjunctive normal form*. The boolean satisfiability problem is decidable, but belongs to the class of NP-complete problems. Thus, algorithms solving this problem are intractable. However, there are specializations of the SAT problem for which a solution can be computed in polynomial time, for example the *horn satisfiability problem* (HORN–SAT). In HORN–SAT the boolean formulas are restricted to horn formulas. A horn formula consists of a conjunction of horn clauses. A horn clause is a disjunctive clause with at most one positive literal. A *literal* can either be a variable (positive literal) or the negation of a variable (negative literal).
DPLL [20] is a classic algorithm solving the boolean satisfiability problem. The DPLL algorithm combines a backtracking search and local consistency checks. In the backtracking step one literal is selected and the problem splits up in two sub–problems, one in which the selected literal is assigned true and one, where it is set to false. All disjunctive clauses that become true by the assignment can be removed from the formula, and from those clauses that do not evaluate to true the assigned literal is removed. When there is a clause consisting of only one literal this literal can only be assigned in one way to make that clause true. In unit propagation this literal will be assigned and all clauses containing

that literal can be removed from the formula. From all clauses containing the opposite literal the literal is removed. Unit propagation is repeated until there are no more clauses containing a single literal. Next, pure literals are eliminated. A pure literal is a literal that is only present with one polarity. When a variable is for example only occurring as negative literal one can assign that variable to false and all clauses containing the negative literal are evaluating to true and can be removed. If in the end there is one empty clause, that is, a clause for which all literals are assigned and evaluated to false, then the current sub–problem is not satisfiable and one has to backtrack. If there are no more clauses left that means for the current assignment of literals every clause contains at least one literal evaluating to true then a satisfying assignment for the variables has been found.

2.2.7 Satisfiability Modulo Theories

An important generalisation of the SAT is the *satisfiability modulo theories* (SMT) problem. Here, instead of literals, sentences can appear that evaluate to true or false in some theory. Common background theories used for SMT are the free theory, linear arithmetic, or the theory of bit vectors. In the equations (2.11)-(2.12) we see an example formula containing propositions in linear arithmetic, integer arithmetic, and one boolean literal.

$$x_1, x_2 \in \mathbb{R}, \quad x_3 \in \mathbb{Z}, \quad x_4 \in \mathbb{B} \tag{2.11}$$

$$(x_1 \leq 5) \wedge (-x_2 \leq 10) \vee (x_3 = 2) \wedge x_4 \tag{2.12}$$

The decidability of SMT problem formulations depends on the background theories used. If the SMT is restricted to a set of decidable background theories then the SMT is decidable. If undecidable theories such as non–linear arithmetic including transcendent functions are allowed then the SMT is undecidable. The well–known SMT solver CVC currently implements a large number of logical theories and their combinations [21].

2.2.8 Properties of the Search Space

Every specialization of the CSP is imposing constraints on the formulation of constraints and on the selection of domains. For the problems presented so far, we focused on the way constraints are stated. In this section we present further problems that can be obtained by strengthening or relaxing the constraints imposed on the selection of domains.

Mixed Integer Programming

Domains can either be continuous or discrete. The optimisation problem as well as linear programming permits only continuous domains. They can be generalised by allowing variables in the discrete integer domain.
Mixed integer linear programming (MILP) is a generalisation of linear programming allowing variables in the integer domain. While linear programming can be solved in polynomial time, the introduction of variables with discrete domains makes the problem NP-complete. A famous instance of this problem is the knapsack problem. In the knapsack problem it is the objective to maximise the value of undividable objects in a container where each object has a value and a weight. The sum of the weights must stay below a threshold. Mixed integer linear programming is decidable.
Mixed integer non–linear programming (MINLP) is a generalisation of the optimisation problem allowing variables in the integer domain as well as the real domain. This problem

contains many instances that are known to be infeasible to solve. A well–known instance is the factorisation of a number with two prime factors. Mixed integer non–linear programming is just as the optimisation problem undecidable. But while to the optimisation problem tractable heuristics similar to the algorithms for the convex optimisation problem can be applied the introduction of discrete variables also makes intractable heuristics necessary to solve mixed integer non–linear programs. An instance of mixed integer non–linear programming is as follows:

$$x_1, x_2 \in \mathbb{Z}, \quad x_3 \in \mathbb{R} \tag{2.13}$$

$$\text{minimise:} \quad x_3 \tag{2.14}$$

$$\text{subject to:} \quad x_1^2 + x_2^2 - 10.5 + x_3 \leq 0 \tag{2.15}$$

$$\text{subject to:} \quad x_3 \geq 0 \tag{2.16}$$

One possible solution to this problem is $x_1 = 1$ and $x_2 = 3$ and the optimal value for x_3 is 0.5. Without the integral constraint on x_1 and x_2 the solution to this problem instance could be computed by an interior point method in polynomial runtime; x_3 would then be 0. With $x_1, x_2 \in \mathbb{N}$ one can estimate an upper and a lower bound for x_1 and x_2 and then needs to try out all combinations of possible values for those two variables.

To solve mixed integer linear programs a combination of different methods is used. One strategy is the cutting–planes method. For cutting–planes first the optimal solution of the *continuous relaxation* is computed. The continuous relaxation of a mixed integer linear program is the linear program where all discrete domains are replaced by corresponding continuous domains. For example, the set $\{1, 2, 3\}$ could be replaced by the interval $[1, 3]$ If the optimal solution is integral in all the components that are required to be an integer we are done. If that is not the case a constraint is added such that all feasible solutions are still within the new feasible set, but the currently optimal solution of the continuous relaxation is excluded. Then, the solution of the continuous relaxation of the new problem is computed. This repeats until no more additional constraint can be found or the solution is integral in those components that are required to be integral. Mathematicians have found a variety of strategies to generate those additional constraints for the cutting–planes method. Another strategy is the branch–and–bound method. In branch–and–bound, the solution of the continuous relaxation is computed first and, in case the integrality constraints are not satisfied, the problem is split into sub–problems excluding the current solution of the continuous relaxation. For example the problem above might be split up into two sub–problems, by one time adding the constraint $x_2 \geq 1$ and one time adding $x_2 \leq 0$ to the original problem. This separation would exclude all solutions with x_2 between 0 and 1. The solution of the continuous relaxation, however, gives a lower bound for the optimal value. An upper bound for the optimal value of a sub–problem can be gained by a heuristic. If the lower bound of sub–problem A is above the upper bound for sub–problem B then sub–problem A can be pruned. Thus, large portions of the search space are excluded from the search.

The same methods can be applied to mixed integer non–linear programming analogous. But for them it is not an algorithm but only a good working heuristic.

Finite Constraint Satisfaction Problem

When each domain is a finite set then the search space is also finite. We call a constraint satisfaction problem with finite search space a *finite constraint satisfaction problem* (FCSP). In fact, on a computer, a real number is stored in IEEE 754 floating point format, which is a finite discretization of the real numbers.

For a *finite constraint satisfaction problem* at least theoretically one can always find a solution or deny the existence of a solution after exhaustive search, thus it is decidable. In practice the search space is finite but often too large for exhaustive search.
Instances of the finite constraint satisfaction problem can be solved by reducing the domains of variables by constraint propagation and checking locally for consistency and backtracking when an inconsistency is found [22]. Constraint propagation is a kind of logical inference and based on rules for reasoning. For example from the two constraints $x \leq 5$ and $y \leq x$ one can conclude that $y \leq 5$, thus the domain of y has been reduced. Examples of working constraint logic programming systems are B-Prolog and SWI-Prolog [23].

2.3 The AMPL Modelling System

'**A** **M**athematical **P**rogramming **L**anguage' (AMPL) from Robert Fourer [24] is a mathematical programming language language and also a software system interfacing different solvers capable of finding valid assignments for variables in an AMPL program, that is, an instance of the constraint satisfaction problem or one of its specializations presented in Section 2.2. An AMPL program consists of two parts: the AMPL model and the AMPL data. The AMPL software system additionally provides a command language allowing the user to interact with the solver.
In this section we will shortly explain the standard work flow with the command language and the concepts of the AMPL model language and the AMPL data language that are needed in this thesis.

2.3.1 Command Language

When AMPL is started in interactive mode it shows a command prompt requiring the user to either enter commands or directly modify the AMPL model. When AMPL is started with an AMPL script in the command line arguments it operates in batch mode. The normal workflow using AMPL is to load an AMPL model from a file or to enter the AMPL model directly in the console. Then one switches into the data entering mode and inputs specific values for every parameter of the loaded AMPL model or directly loads the data from a file. Now the AMPL program is ready to be solved by one of the solvers interfacing with AMPL. Therefore, we need to tell AMPL which solver shall be used and possibly set some further options. Then the `solve` command is issued and AMPL will return as soon as either a solution is found, or an error has occurred, or the AMPL program has been found to be infeasible. When AMPL returns from the solve process reporting that a solution has been found, one will use the `show` command to print the satisfying variable assignments. Furthermore, AMPL provides commands to reset the currently loaded data or data and model. Commands for making on the fly modifications to the AMPL model and data without a complete reset are also available. For detailed reference of the commands the AMPL system supports, we refer to the AMPL reference manual in the appendix of [24].

2.3.2 Modelling Language

The AMPL modelling language is the means used to formalise parameterised constraint satisfaction problems. The modelling language supports several model entities. The supported model entities relevant for our thesis are sets, parameters, variables, constraints, and objectives. Further we are using set expressions, index expressions, and arithmetic or logical expressions. We will start by explaining the expression types and then introduce

the aforementioned model entities in the given order. For detailed reference we refer to [24].

Expressions

Set and Index Expressions A set expression specifies a collection of values. The easiest way to specify a set is by explicitly naming each element, for example, `1 2 3 4` specifies the set $\{1, 2, 3, 4\}$. A set containing a range of integers can also be specified more efficiently by stating `1..4`. AMPL supports some set operations allowing one to specify the union or intersection of two set expressions.
Another important expression using the set expressions are index expressions. An index expression consists of an index variable and a set expression specifying the values that are assigned to the index variable one after another. Index expressions are used to iterate through a collection. An index expression in AMPL looks like `{i in 0..4}` where `i` is the index variable, which takes any integer value from 0 to 4.

Arithmetic, Logic, and Relation Expressions In AMPL arithmetic and logical expressions can be composed of a variety of arithmetic functions, constant values, variables, and parameters. AMPL has built in the basic arithmetic logical operations such as `+`,`-`,`*`,`/`,`and`,`or`,`not` and also the exponentiation and further additional functions like `sin`, `cos`, `if--then--else`. The set of functions available in AMPL can be extended further by loading external function libraries. The syntax of the arithmetic, logic and relation expressions in the AMPL modelling language is very intuitive. Examples of valid expressions evaluating to true or false are: `x<5; a=sin(b); (a=5) and (not (b<=sin(c)))`. Expressions returning a numerical value are: `x`; `if (x<=5) then (10) else (11)`; `cos(b)`; `max(a,b,10)`. We assume the used literals `x`, `a`, `b` and `c` to be names of declared variables or parameters.

Sets

A set is a container holding a collection of values. It is declared with the keyword `set`, it has a name and attributes. A set can have several dimensions, be ordered or unordered, and the default content can be specified. The content of the set can be required to be a subset of some set expression. A set expression is used to specify the elements contained in a set. The content of a set can be directly specified in the model or data section. When the content of a certain set is specified in the data then the model only contains a declaration of the set.
The simplest set declaration would be: `set A;` just declaring a one dimensional set named `A` whose contents have to be specified in the data section. `set A within 0..4 default 1 3;` declares the set `A` whose content need to be a subset of $\{0, 1, 2, 3, 4\}$ and by default, when nothing else is specified in the data, the set will hold the elements 1 and 3.

Parameters

A parameter holds a numerical value that can be accessed anywhere in the model. It is fixed and its value is usually specified in the data section. A basic parameter declaration looks like this: `param b;`. This expression specifies a parameter b that can be used anywhere in the model. More sophisticated features of parameters will not be used in this thesis.

Variables

A variable in AMPL has an identifier and holds a numerical value. It can either be in a discrete or continuous domain, and it can have an upper and a lower bound. A variable with a continuous domain is declared by the statement: `var x;`. A variable with discrete domain is declared by the statement: `var x integer;`. Any model entity can be declared as an indexed collection by appending an index expression to the declaration. A variable being declared as an indexed collection of variables will look like this: `var x{i in 0..4};`. This statement essentially declares 5 different variables namely `x[0]`, ..., `x[4]`. Variables can, just like parameters, be referenced anywhere in the AMPL model. When a solver is called the solver will try to assign the values of the variables in such a way that all constraints in the model evaluate to true and the objective is optimised.

Constraints

A constraint in AMPL has a name and an expression evaluating to either true or false. As already explained for variables also constraints can be declared as an indexed collection of constraints. The index variable can also be accessed inside of an indexed constraint. The declaration `such that MyConstraint{i in 0..4}: x[i] <= i*b;` declares 5 constraints named myConstraint[0], ..., myConstraint[4]. We assume that b is the parameter and x the indexed collection of variables we have declared in the last two paragraphs. Now each of the 5 constraints refers to another index of x and imposes an upper bound on it, which also depends on the index. We could also have explicitly written out all 5 constraints.

Objectives

An objective can either be the maximisation or the minimisation of an objective function. The objective in AMPL has a name and an expression evaluating to a numerical value and specifies whether to minimise or maximise the given expression. The declaration `minimize totalCost: a + b + c;` specifies that the totalCost has to be minimised and tells us that the totalCost composes from the values a, b and c which can be either variables or parameters.

2.3.3 Entering Data

Although it is possible to specify the values of parameters as well as the elements contained in the declared sets directly in the AMPL model it is often desirable to enter them separately as data. Data can be entered in two ways either by reading them from a file or by switching into the data mode and entering it on the console. We assume that in the model the parameter `b` and the set `A` have been declared. Entering `param b := 5;` and `set A := 0 2 5;` in the data mode would set `b` to 5 and set the content of `A` to $\{0, 2, 5\}$.

3 Generating Unit Tests from a UML Activity

In this chapter we will present an algorithm that generates unit tests for C functions from their **U**nified **M**odeling **L**anguage™ (UML) activity model. The test generation process is divided in five steps. Each step can be seen as model–to–model, model–to–text, or text–to–model transformation with the input and output language or model specified. The architecture makes it particularly easy to replace any of the five steps or extend it with additional features. We do not only specify an algorithm for generating unit tests from a UML model but also a framework for building further algorithms that generate unit tests from arbitrary input models. If, for instance, another input language specifying a control flow graph will be used then only the first step would need a few changes. In some steps we will even specify alternative transformations.
Our approach for automated generation of test data is based on mathematical programming and the approach for finding relevant control flow paths is based on a breadth first search with early infeasible path recognition.

3.1 General Overview of the Workflow

The transformation from an UML activity diagram to a C++ unit test code is divided in five steps. Those five steps are: Normalisation, Rigorous Mathematical Programming, Abstract Test Case Generation, Specific Test Data Generation, and finally Unit Test Synthesis. Here we want to give a brief overview of those five steps. Each of these steps as well as the interfaces between them will be described in detail in the following sections.
In the first step, Normalisation, we check some design rules, parse all embedded OCL constraints, and map the relevant parts from the UML model onto a simplified meta model of an activity. Throughout the next step we generate a mathematical program out of our simplified test model. We use the AMPL language for specifying a rigorous mathematical program. The third step, Abstract Test Case Generation, is a path search where we can find all control flow paths, or those that are necessary to fulfil some coverage criterion of choice. We call a control flow path from an initial node to a final node of an activity *abstract test case*. In the fourth step, Specific Test Data Generation, we have each abstract test case encoded as input data for the AMPL model generated during the second step. The AMPL model will be solved by a state–of–the–art solver that interfaces with AMPL. For each abstract test case the solver was able to find valid test data we will store the data in a *test case model*. This test case model serves as input for the last step. During Unit Test Synthesis, we take the generated solution of the mathematical program and put the values in place within compilable and executable C++ unit test code. An overview of the complete workflow of our approach is given in Figure 3.1. There we see an activity diagram showing the five steps of our algorithm and within each step the sub–tasks are visible. Each of the depicted central buffer nodes will hold the output of one step of the algorithm. The labelled input pins at the actions indicate which task accesses which previously generated item.

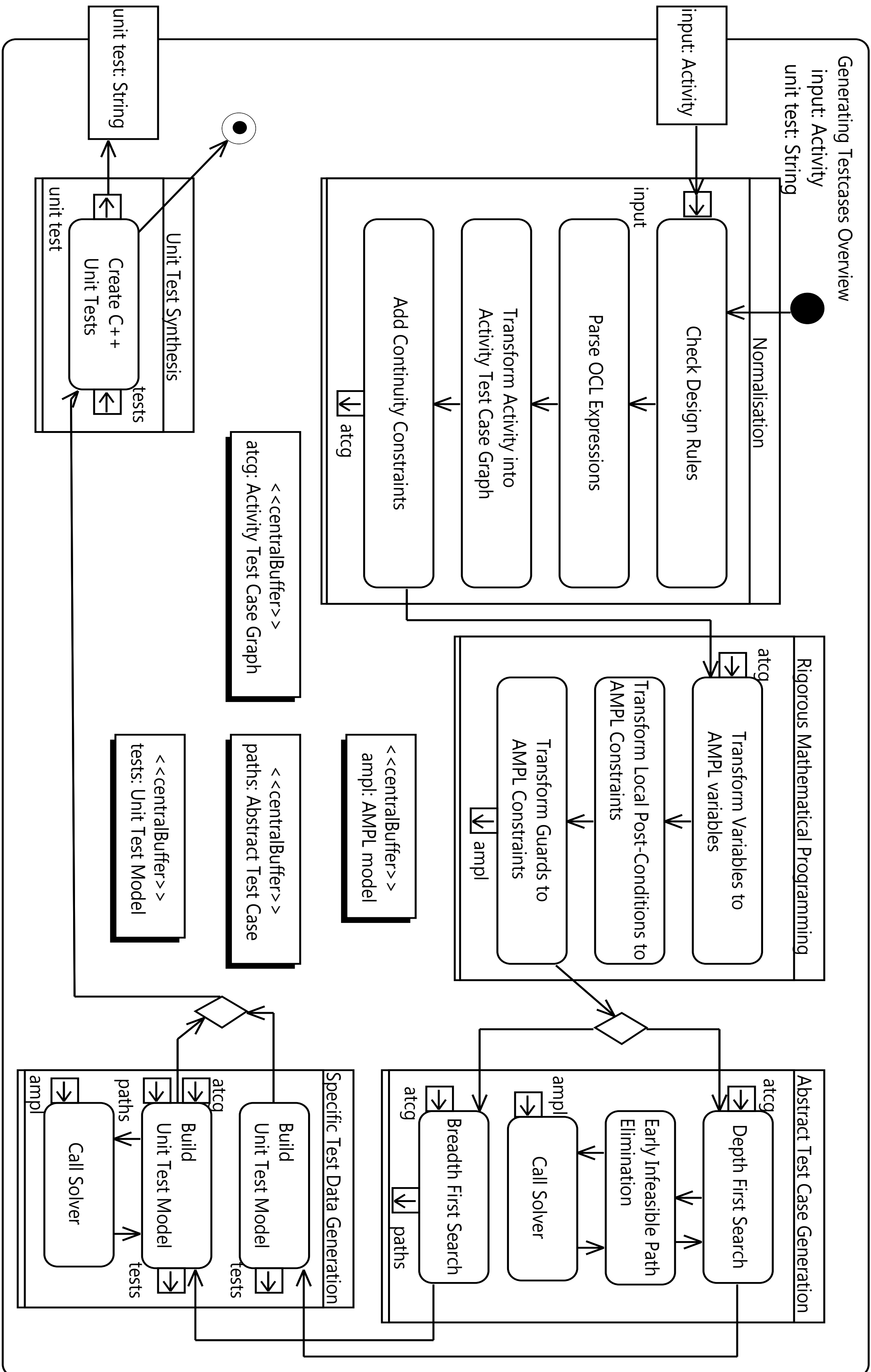

Figure 3.1: Overview of the workflow for unit test generation

We want to outline two special features of our algorithm. Early infeasible path recognition as described in Section 3.4.3 enables us to recognise sub–paths for which no valid input data exists during the path search in the third step. When early infeasible path recognition is used the third and fourth step are interfering with each other. In this case test data is already generated in the Abstract Test Case Generation. Another feature of the specified algorithm is that it generates boundary values for each abstract test case. For each control flow path in an activity there is a set of input values that will satisfy all constraints along this path and thus presents suitable test data for this abstract test case. In testing, bugs often are likely to be triggered when a point at the edge of this set is selected as test data. How we are assuring that a boundary value will be found during test data generation will be explained in Section 3.5.6.

3.2 Normalisation

The **U**nified **M**odeling **L**anguage™ (UML) specifies about 250 modelling elements each having several references and attributes. An algorithm interpreting every element that is defined in the UML becomes overly complex. We interpret only a subset of the UML consisting of 16 different modelling elements. For a complete reference of the UML elements we refer to the UML2.3 superstructure specification [25]. In industrial applications a model will usually contain the complete specification for a product consisting of thousands of objects and containing elements we are not interested in. Through the references of any object within a model we can navigate to any other object within the model. In order to not get lost in such a model when generating unit tests from it, we need to slice out the relevant sub–model and perform a pre–processing step to ensure our algorithm does not run into error conditions. Especially the contained OCL expressions need to be parsed and checked for conformity with the subset of OCL our algorithm is able to handle correctly. The input of the Normalisation step is an UML *Activity* from which we traverse all relevant attributes and references and store each bit of information needed in consequent steps in our own unambiguous intermediate representation. The input model can be discarded after this step. All consequent steps we will exclusively use our own intermediate representation and no more access the original UML model. Throughout this section and the rest of this thesis we will print specific modelling elements and associations of the UML in a special font, for example *Activity* and *ownedNodes*, to make it clear that the UML modelling element or the UML association as described in the UML specification [25] is meant.

3.2.1 Design Rules for the Test Model

Structural Design Rules

For our test generation we assume the models to comply to certain design rules. The input *Activity* is an *ownedActivity* of a *Class*. The *Class* containing the *Activity* itself is contained by a *Package*. There is an *Operation* specifying the *Activity*. This specifying *Operation* is either a sibling in the UML tree structure and has the same name as the *Activity* or is explicitly specified by the *specification* reference of the *Activity*. The specifying *Operation* will be needed when parsing OCL constraints in Section 3.2.3. In Figure 3.2 we see a tree view of an UML model where those requirements are met.

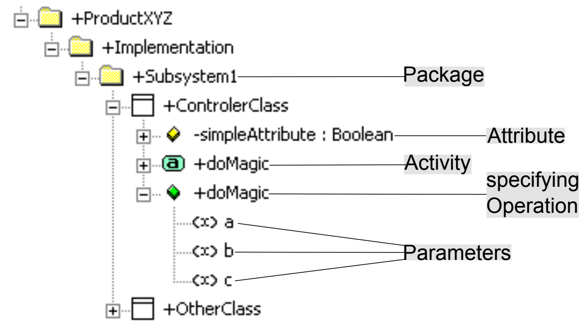

Figure 3.2: Screenshot of a valid model in the model browser of Atego® Artisan Studio

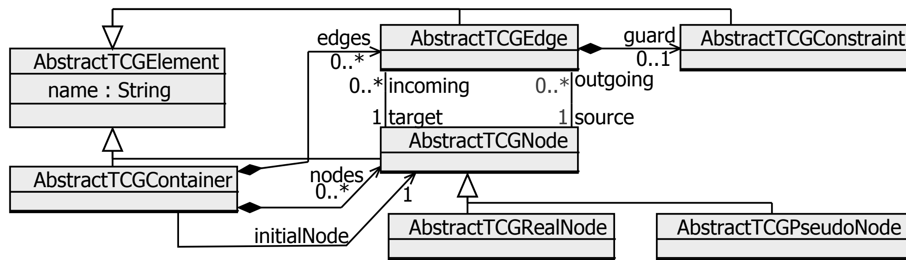

Figure 3.3: Meta model of the abstract test case graph

OCL design rules

Further we assume that the embedded textual OCL constraints are either contained within a *LiteralString* element or an *OpaqueExpression* with the according *language* value set to 'OCL'. We support a strict subset of the OCL language. The Extended Backus–Naur Form (EBNF) of the supported OCL subset is shown in Algorithm 1. Any parsed OCL expression must be an instance of the <Bool> production rule.

3.2.2 A Meta Model Suitable for Automated Unit Test Generation

The developed unit test generation algorithm does not work directly on an UML model but on an *activity test case graph*. The activity test case graph contains only those details from the UML meta model that are really necessary for the unit test generation. Since it contains only parsed OCL expressions as abstract syntax trees it is much more suitable for the transformation into a mathematical program.
The activity test case graph is defined as an extension of a more general *abstract test case graph* model. The abstract test case graph meta model was tailored to fit activity diagrams as well as UML state machine diagrams in order to be able to apply algorithms not only to activity diagrams but also to state machine diagrams. This common abstract meta model is also a provision to be able to reuse existing test case generation algorithms from ParTeG [10] for *Activities*. The Eclipse plug–in developed along with this thesis will be integrated with ParTeG. ParTeG is a test generator for activity diagrams with advanced support for different coverage criteria.

Algorithm 1 Extended Backus–Naur Form of supported OCL subset

<Bool> ::= <LogicalOperation> | <RelationOperation> | <BooleanLiteral> | <BooleanVariable> ;

<LogicalOperation> ::= <Bool>, <LogicalOpSymbol>, <Bool> ;

<LogicalOpSymbol> ::= "and" | "or" ;

<RelationOperation> ::= <Number>, <RelationOpSymbol>, <Number> ;

<RelationOpSymbol> ::= "<" | ">" | "<=" | ">=" | "=" | "<>" ;

<Number> ::= <ArithmeticOperation> | <IntegerLiteral> | <RealLiteral> | <IntVariable> | <RealVariable> ;

<ArithmeticOperation> ::= <Number>, <ArithmeticOpSymbol>, <Number> ;

<ArithmeticOpSymbol> ::= "+" | "-" | "*" | "/" ;

<BooleanLiteral> ::= "true" | "false" ;

<IntegerLiteral> ::= ["-" | "+"], <Digit>, {<Digit>} ;

<RealLiteral> ::= ["-" | "+"], <Digit>, {<Digit>}, ".", {<Digit>} ;

<Digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

<BooleanVariable> ::= ?OCL expression evaluating to a *Property* or *Parameter* of type boolean? ;

<IntegerVariable> ::= ?OCL expression evaluating to a *Property* or *Parameter* of type integer? ;

<RealVariable> ::= ?OCL expression evaluating to a *Property* or *Parameter* of type real? ;

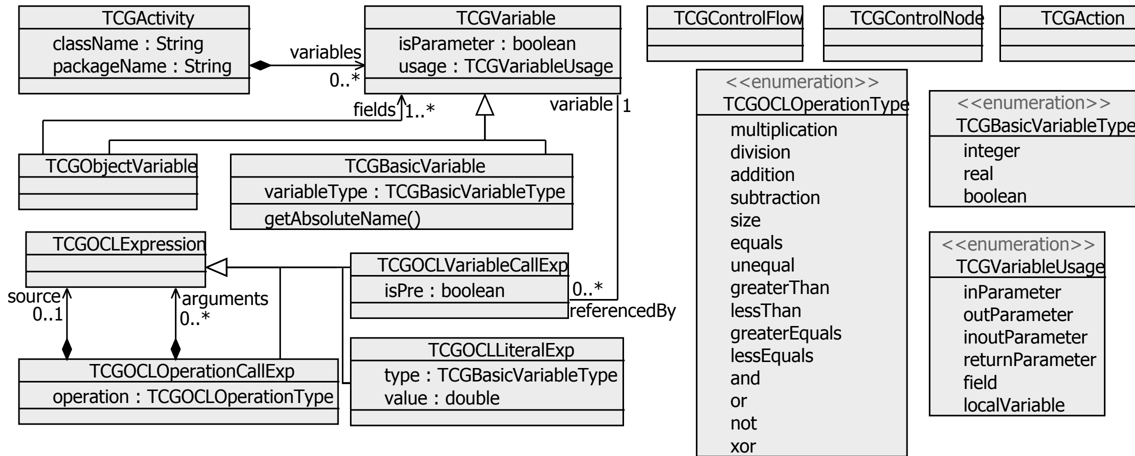

Figure 3.4: Meta model of the activity test case graph

Abstract Test Case Graph

An abstract test case graph is a directed graph consisting of nodes (AbstractTCGNode) and edges (AbstractTCGEdge). Each edge has a source and a target node. Each node can have multiple outgoing and multiple incoming edges. A node can be either a pseudo node (AbstractTCGPseudoNode) or a real node (AbstractTCGRealNode). A pseudo node is a node that can be inserted in the middle of an edge without changing the semantics of the test model. Edges can have a guard condition of type AbstractTCGConstraint. Such a graph is contained by a container (AbstractTCGContainer). The container has a singular initialNode reference to one of its nodes exposing this as the initial node. The class diagram of the abstract test case graph meta model is shown in Figure 3.3.
The semantic of the activity test case graph is like a Petri net. When executing an abstract test case graph we have at the beginning a token in the initial node that can move along the enabled edges. An edge is enabled when its guard condition is true. We say that a node is being executed when a token resides in the node.

Activity Test Case Graph

An activity test case graph is an extension to the abstract test case graph especially tailored for test generation from *Activities*. The activity test case graph models control flow and constraints on variables. The constraints are required to hold at different points during execution of the activity test case graph. Its meta model is shown in Figure 3.4.
The TCGActivity is an extension of the AbstractTCGContainer. TCGAction, TCGControlNode, and TCGControlFlow refine the types AbstractTCGRealNode, AbstractTCGPseudoNode, and AbstractTCGEdge respectively. The main extensions to the abstract test case graph meta model are the variables (TCGVariable), the elements of an abstract syntax tree (TCGOCLExpression) to express constraints, and the TCGAction.

TCGAction A TCGAction can, in addition to its super–type, contain arbitrary many localPostconditions of the type AbstractTCGConstraint. A local post–condition has the semantics that it has to be true after the execution of the action. This implies potentially changing some variables throughout the execution of the action.

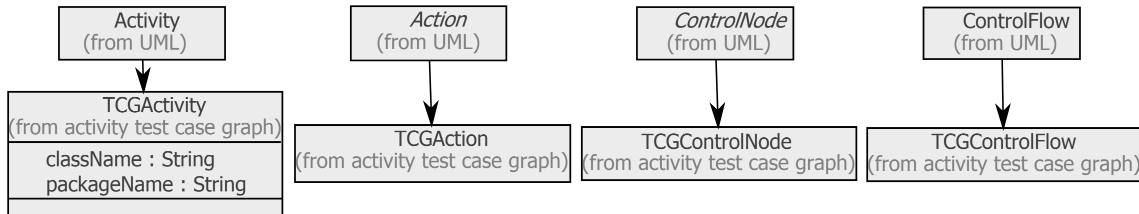

Figure 3.5: Straightforward mapping from UML elements to activity test case graph elements

TCGOCLExpression TCGOCLExpression is a subtype of AbstractTCGConstraint. A TCGOCLExpression can either be a TCGOCLOperation call with a source and arguments, or a TCGOCLLiteralExpression holding a literal value and its data type, or a TCGOCLVariableCallExp referencing a TCGVariable. The isPre attribute of TCGOCLVariableCallExp denotes whether the original OCL expression contained an `@pre` token. TCGOCLExpression and its subtypes form a simplified OCL abstract syntax tree.

TCGVariable A TCGVariable can be one of two subtypes: either a TCGBasicVariable or a TCGObjectVariable. An object variable is put together by one or more other variables. A basic variable has a variableType attribute holding a TCGBasicVariableType literal. The TCGBasicVariableType literal represents one of the types that our algorithm can handle: integer, real, and boolean. A TCGVariable is a placeholder for a value. When the isParameter attribute is true it can only hold one value throughout the execution of the TCGActivity, otherwise it can change its value during each execution of a TCGAction.

3.2.3 Transforming an UML Activity Diagram to an Activity Test Case Graph

An instance of the activity test case graph meta model serves as a normalised input model. We use a model–to–model transformation to transform an *Activity* from a UML model into an activity test case graph. In many cases the transformation is straightforward: for one element in the UML model the corresponding element in the activity test case graph is produced. When one element of the UML model is mapped to one element of the activity test case graph model we call the UML element source element and the created element in the activity test case graph model target element. The straight forward mappings from one source element to one target element are shown in Figure 3.5.

Mapping UML Elements to Activity Test Case Graph Elements

Activity A UML *Activity* is transformed into a TCGActivity. For the C++ unit test synthesis described in Section 3.6 the name of the containing *Class* of the source element as well as the full pathname to the *Package* containing this *Class* need to be stored. The class name and the full pathname will be stored in the property fields of the TCGActivity. For each *Activity* all of its *ownedNodes* as well as *ownedEdges* are considered for transformation.

Action *OwnedNodes* of an *Activity* that are a subtype of *Action* are transformed into a TCGAction. The *Action*'s *localPostcondition* reference may contain textual OCL that will be parsed. The detailed semantics of special subtypes of an *Action* is neglected.

ControlNode *OwnedNodes* of the *Activity* that are a subtype of *ControlNode* are transformed into TCGControlNodes. That means they are pseudo nodes. It is assumed that there is only one UML *InitialNode* whose target element is referenced by the TCGActivity as initialNode.

ControlFlow Based on the *ownedEdges* association of the *Activity* we find *ControlFlow*s that are transformed into TCGControlFlows. If there is a *ValueSpecification* contained in the *guard*, the potentially contained textual OCL needs to be parsed.

Parsing OCL Constraints

*Constraint*s can be found in the *localPostcondition* reference of an *Action*. Each *Constraint* holds a *ValueSpecification* in its *specification* reference. A *ControlFlow* can hold a *ValueSpecification* in its *guard* reference.
Constraint, *ValueSpecification*, and *Property* as well as *Parameter* can not be transformed straightforwardly. The transformation will be done in three steps. First, textual OCL expressions will be extracted from *ValueSpecification*s and then the textual OCL expression will be parsed. When the OCL expression was parsed correctly the parser returns an abstract syntax tree of this expression. In the third step the elements of this abstract syntax tree will be transformed to TCGOCLExpressions and all *Property* and *Parameter* elements referenced by the OCL abstract syntax tree will be transformed into TCGVariables.
We ensure that only those OCL expressions are in the activity test case graph that later on can be transformed into an AMPL model as explained in in Section 3.3. There might for example be *ownedAttribute*s of a *Class* that are not changed by any of the transformed *Actions* and whose value do not have any influence on any of the transformed *ControlFlow*'s *guard*. The activity test case graph will not contain any TCGVariable representing such an irrelevant *Property* or *Parameter*.

Extracting Textual OCL *LiteralString* and *OpaqueExpression* are subtypes of *ValueSpecification* potentially containing textual OCL expressions. For a *LiteralString* we will try to parse its *value* as OCL. For an *OpaqueExpression* we will first check whether the *language* attribute contains the value 'OCL' and try to parse the corresponding value of the *body* attribute. If the *language* attribute does not contain the value 'OCL' then we will try to parse the concatenation of all *body*s of the *OpaqueExpression*.

Parsing Textual OCL For details of the **O**bject **C**onstraint **L**anguage (OCL) we refer to the OCL 2.3 specification [26]. For this thesis we consider three types of constraints: invariants, post–conditions and guard conditions. Every OCL constraint is parsed with respect to a context. The context specifies the evaluation of the `self` keyword and which other variables and parameters are accessible from the OCL expression.
Textual OCL found inside a *Constraint* contained by the *ownedRule* reference of either the *Activity*, or its containing *Class* will be interpreted as invariant. Invariants need to be true before the *Activity* is executed and after the execution of the *Activity* has finished. The OCL expressions found within *guards* of *ControlFlows* are required to evaluate to true, whenever the *ControlFlow* is traversed. We are parsing the invariants as well as guards according to the OCL specification as invariants of the specifying *Operation*.
Actions can contain multiple *Constraints* within their *localPostcondition* reference. Every post–conditions is required to hold after the execution of the *Action* containing it. Only in post–conditions the OCL `@pre` is allowed. With `@pre` we refer to the value of a variable

before the execution of the *Action*. Any textual OCL found within a *localPostcondition* will be parsed as post–condition in the context of the specifying *Operation*. That means for every parsed OCL expression all attributes of the *Class* as well as all *ownedParameters* of the specifying *Operation*, and all properties of the containing *Packages* are accessible, but only in local post–conditions the `@pre` is valid.

Transforming the Abstract Syntax Tree If the OCL parser was supplied with a valid OCL expression for the context that was assumed then it will return an abstract syntax tree for the parsed OCL expression. We assume the parsed OCL complied with the rules explained in Section 3.2.1 and the OCL abstract syntax tree returned consists of those tokens specified in Algorithm 1.
<BooleanLiteral>, <IntegerLiteral>, and <RealLiteral> are transformed into a TCGOCLLiteralExp. Its value is stored in double precision floating point format. The type attribute of the target element is set according to the type of the source element. For the type boolean a value of 0.0 means false and 1.0 means a true.
<BooleanVariable>, <IntegerVariable>, and <RealVariable> tokens are transformed into TCGOCLVariableCallExp elements. If the original OCL token was tagged with an `@pre` then the isPre attribute is set to true otherwise it is false. <BooleanVariable>, <IntegerVariable>, and <RealVariable> elements represent an arbitrary OCL expression that evaluates either to a *Property* or a *Parameter*. The *Property* or *Parameter* pointed to by the expression will be transformed into a TCGVariable and the variable reference of the TCGOCLVariableCallExp will be set to point to this TCGVariable.
We transform a <LogicalOpSymbol>, <RelationOpSymbol>, or <ArithmeticOpSymbol> into the enumeration literal of the TCGOCLOperationType enumeration that matches the token represented by the source element. The token `=` matches for example the equals enumeration literal.
Each <LogicalOperation>, <ArithmeticOperation>, and <RelationalOperation> will be transformed into a TCGOCLOperationCallExp. The value of the operation attribute will be obtained by transforming the <RelationalOpSymbol>, <LogicalOpSymbol>, or <ArithmeticOpSymbol> contained in the source element. The target element of the first operand of each operation will be stored in the source reference of the TCGOCLOperation. The target element of the second operand will be stored in the arguments reference of the TCGOCLOperationCallExp.
The productions <Bool> and <Number> in Algorithm 1 are there to keep the EBNF notation more readable. No TCGOCLExpression element is generated for them but the transformation of a <Bool> or <Number> element will transform the contained abstract syntax tree node and return the result of this transformation.
Every root TCGOCLExpression directly contained by a TCGAction, TCGControlFlow, or TCGActivity gets its name property set. The transformation ensures that every root TCGOCLConstraint has a globally unique name. In the original UML model it is not the case that each *Constraint* or *ValueSpecification* has a globally unique *name*.

Transforming Properties and Parameters UML *Property* or *Parameter* elements referenced by a <BooleanVariable>, <IntegerVariable>, or <RealVariable> element will be transformed into a TCGVariable. All TCGVariables are contained by the variables reference of the root element of the activity graph, the TCGActivity.
For *Parameter* and *Property* elements we need to check their *type* reference. If the *name* of the *Type* is one out of a list of names that can be mapped to a literal of the TCGBasicVariableType enumeration then a TCGBasicVariable is created for it and its variableType

field set accordingly to the enumeration literal that is either integer, real, or boolean. The exact mapping of *name*s of the *Type* to a TCGvariableType is implementation specific. One could for example map from a *Type* named 'uint32_t' to integer.
If a *Parameter* was transformed into a TCGBasicVariable then the isParameter field is set to true, indicating that this variable has been a *Parameter*. For every *Property* the isParameter field of the target element will be set to false.
The field usage of the TCGBasicVariable can take values from the enumeration TCGVariableUsage. This information will be needed when creating the C or C++ unit tests as described in Section 3.6. If the TCGBasicVariable was created to represent a *Parameter* then its usage can be inParameter, outParameter, inoutParameter, or returnParameter. The value of the usage field for a transformed *Parameter* is determined by the value of the *direction* field of the original *Parameter*. If the source element of a TCGBasicVariable was a *Property* then its usage is field.

3.2.4 Adding Continuity Constraints

From imperative programming we are used that variables can only change their value when this is explicitly stated in an assignment. It is especially important that x will not take a new value during a computation if we do not explicitly state an assignment for x. In OCL there is nothing like an assignment. In OCL we express constraints specifying which assignment would be legal and which not. If for a specific variable x there is no constraint referring to it that means that this variable can take any value from its domain. Consequently the user needs to make it explicit when a variable shall not be changed during the execution of an action. The user can state that a variable x shall stay unchanged by adding a *continuity constraint* to the post–condition of each action that shall not change the variable. The continuity constraint will have the semantic of the textual OCL `x=x@pre`. That means x shall have the same value as x had before the execution of this action.
We can support the user by adding a large amount of continuity constraints automatically to the activity test case graph. All variables that are not constrained at all within the post–condition of an action will have the same value after execution of the action as before. For each TCGAction within the TCGActivity we need to determine the set of all TCGVariables in the TCGActivity that are not referenced from the local post–condition of this TCGAction. This is the set of variables that need a continuity constraint to prevent the constraint solver from setting them to completely arbitrary values. The algorithm for adding continuity constraints is printed in Algorithm 2. The isContainedBy() function used in line 2 is true if the self element is directly or indirectly contained by the argument. The navigated referencedBy reference is the opposite of the TCGOCLVariableCallExp's variable reference.

Algorithm 2 Algorithm for adding continuity constraints

Require: atcg := A TCGActivity
Ensure: All necessary continuity constraints are added for each TCGAction in atcg
1: **for** action : TCGAction ∈ atcg.nodes **do**
2: S ← atcg.variables->reject(var : TCGVariable | var.referenceBy->exists(exp : TCGOCLVariableCallExp | exp.isPre = false and exp.isContainedBy(action)))
3: **for** var : TCGVariable ∈ S **do**
4: Add continuity constraint for var to action.localPostconditions
5: **end for**
6: **end for**

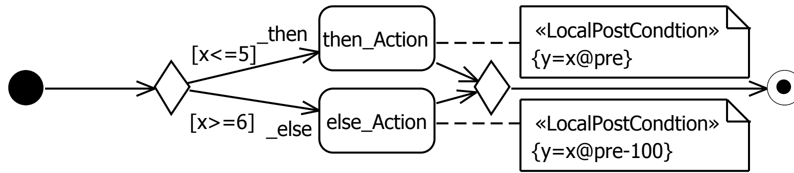

Figure 3.6: Example of a simple activity with two actions and one decision

3.3 Rigorous Mathematical Programming

In order to make the model executable we need a rigorous mathematical representation for it. We will formulate the problem of finding appropriate test data for a certain control flow path in the activity test case graph as instance of one of the problems presented in Section 2.2. Which problem that will be an instance of, depends on the input model.
In the previous section we have extracted all necessary information from a UML model and stored it in an activity test case graph model. All OCL expressions have been parsed and stored as abstract syntax tree in the activity test case graph model. In this step we will exclusively use the activity test case graph as source model and no longer access the original UML model. We will perform a model–to–text transformation from an activity test case graph to an '**A** **M**athematical **P**rogramming **L**anguage' (AMPL) program. We explained the AMPL mathematical programming language in Section 2.3. The mathematical program in AMPL consists of two parts: the model and the data. When we want to generate suitable test data for a certain control flow path within an activity test case graph we need an AMPL model and the corresponding AMPL data. An AMPL model encodes a complete activity test case graph including all variables and all constraints that are contained. The control flow path within this activity test case graph is encoded in the AMPL data.
In Section 3.3.1 we will present a small example, where a linear program is built from a simple activity diagram. Then we will explain in detail the generation of the AMPL model in Section 3.3.2-3.3.5, and in Section 3.3.6 we show how the AMPL data for a control flow path is generated.

3.3.1 Introductory Example

During the execution of an activity test case graph we have an initial state where every variable has a specified value. A state is specified by the complete value assignment for all variables. After each execution of an action this state might change according to the rules given in the action's post–conditions. Post–conditions can specify a relation between the value assignment in the current state and a value assignment in the previous state. The states are interconnected with each other via post–conditions. The set of all post–conditions can be seen as a state transition function. In order to traverse a control flow edge its guard needs to evaluate to true in the current state. Guard conditions can only specify relations between the value assignments within a single state.
The AMPL program models the execution of an activity as a series of states. All post–conditions and guards are contained in the model as constraints, which can be switched on and off for each state within the series of states. Assume an activity modelling the statement `if (x<=5) then y=x else y=x-100`. The corresponding activity diagram is shown in Figure 3.6. There are two possible control flow paths. Each possible control flow path contains exactly one action. Consequently we need two states: one initial state and

one state after the execution of the first action. There are two variables, x and y. For each variable we need in AMPL an array of length 2 in order to hold the value assignment for those variables in each state. We will denote the value assignment for x in the i-th state as x_i.
Let us assume we want to generate test data for the `_then` path in the diagram. In this case in the initial state the constraint $x_0 \leq 5$ is activated and in the first state we have the constraint $y_1 = x_0$ activated. Further, as explained in Section 3.2.4, the continuity constraint $x_0 = x_1$ has been introduced automatically since there is no other rule constraining x_1 and we want to preserve the variable's value from the previous state when possible. The resulting problem is a linear program with four unknown values and three linear equations and inequalities. The resulting linear program for this case is shown in equations (3.1)-(3.2). The linear equations do only represent those constraints that are active for the `_then` control flow path in the diagram. The constraints $x_0 \geq 6$ and $y_1 = x_0 - 100$ are not on the `_then` control flow path. The value of y_0 will stay unconstrained, since there is no previous state before the initial state. One can recognise already from this small example that there are quite some 0 entries in the matrices. For larger problems the matrices tend to become sparse. Many of the solvers we present in the Section 3.5.4 will take advantage of sparse matrices when solving the problem.

$$\begin{pmatrix} 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ y_0 \\ y_1 \end{pmatrix} = \vec{0} \tag{3.1}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ y_0 \\ y_1 \end{pmatrix} \leq \begin{pmatrix} 5 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.2}$$

3.3.2 Transforming a TCGActivity to an AMPL Model

Each AMPL model has one parameter called `pathlength` representing the number of actions on a path. For each action with post–conditions and each control flow with a guard there is one *activation set*. An activation set is required to be a subset of [1..*pathlength*] and specifies those states, in which a post–condition or guard condition is active. The value of the parameter `pathlength` as well as the elements of each activation set will be specified in the AMPL data. In this way there is one AMPL model per activity diagram and any control flow path within the activity diagram can be specified in the AMPL data. The AMPL model built will be synthesised as specified in Algorithm 3. We use the 'MOF Model–To–Text Transformation' (MOFM2T) language to specify model–to–text transformation. For each variable in the source model there will be one variable declaration, for each action there will be an activation set declared and multiple sets of constraints. For each control flow having a guard condition there will be an activation set and one set of constraints. The synthesis of AMPL code for variables, actions, and guards will be explained in more detail in Section 3.3.3-3.3.5 respectively.

3.3.3 Transforming TCGVariables

The 'MOF Model–To–Text Transformation' (MOFM2T) template to transform a TCG-Variable into an AMPL variable declaration is printed in Algorithm 4. Every variable has one out of three possible types: integer, real and boolean. In AMPL, by default, a variable

Algorithm 3 Model–to–text transformation from TCGActivity to AMPL model

```
1  [template public ActivityToAMPL(act : TCGActivity)]
2    param pathlength;
3    [for(var : TCGVariable | act.variables)]
4      [VariableToAMPL(var)/]
5    [/for]
6    [for(action : TCGAction | act.nodes)]
7      [ActionToAMPL(action)/]
8    [/for]
9    [for(cf : TCGControlFlow | act.edges)]
10     [ControlFlowToAMPL(cf)/]
11   [/for]
12 [/template]
```

Algorithm 4 Model–to–text transformation from TCGVariable to AMPL model

```
1  [template public VariableToAMPL(var : TCGVariable)]
2    var [var.name/] [if var.isParameter = false] {0..pathlength} [/
       if] [TypeSpecification(var.type)/] :=1;
3  [/template]
4
5  [template public Type2AMPL(type : TCGBasicVariableType)]
6    [if type = integer] : integer >=-10000, <= 10000 [/if]
7    [if type = boolean] in 0..1 [/if]
8    [if type = real] >=-10000, <= 10000 [/if]
9  [/template]
```

is in the domain of real numbers. AMPL also supports variables in the domain of integers. For integers and real numbers we always need an upper and a lower bound that will be set to +10000 and −10000 by default. Without any bounds at all some solvers can not to produce correct results. The boolean domain is modelled by the set expression [0..1], where 0 corresponds to false.
Since a variable can have different values in each state of the execution, each variable will be modelled in AMPL as a variable array of fixed size. The size of the array is defined by the `pathlength`. When the isParameter attribute of the TCGVariable is true then the variable is constant and cannot change its value from one state to another, thus we can model those variables as a single AMPL variable.

3.3.4 Transforming LocalPostConditions

Algorithm 5 Model–to–text transformation from TCGAction to AMPL model

```
1  [template public ActionToAMPL(action : TCGAction)]
2    set [action.name/] within {0..pathlength} default {};
3      [for(constraint : TCGOCLExpression | action.
          localPostconditions)]
4      s.t. [action.name/]_post_[action.localPostconditions->indexOf
          (constraint)/] {i in action.name} :
5      [ConstraintToAMPL(constraint)/] ;
6    [/for]
7  [/template]
8
9  [template public ConstraintToAMPL(constraint : TCGOCLExpression)]
10   [let func : TCGOCLOperationCallExp]
11     ([ConstraintToAMPL(func.source)/] [func.operation.toOpSymbol
          ()/] [ConstraintToAMPL(func.arguments)/])
12   [/let]
13   [let literal : TCGOCLLiteralExp]
14     ([literal.value/])
15   [/let]
16   [let var : TCGOCLVariableCallExp]
17     ([var.variable.name/] [if var.variable.isParameter = false ] [
          i [if var.isIsPre] -1 [/if] ] [/if])
18   [/let]
19 [/template]
```

For each TCGAction the activation set is declared as a subset of [0..`pathlength`] in the AMPL model. The name of the activation set is the name of the TCGAction. The name of each TCGAction is guaranteed to be unique within one activity test case graph model, so name clashes in the AMPL model are prevented (see Section 3.2.3). Each local post–condition of an action is transformed into an indexed collection of constraints over the activation set.
In Algorithm 5 the transformation of a TCGAction into AMPL code is specified in the MOFM2T language. The ActionToAMPL template first generates the declaration of the activation set, see in line 2. If no entries are added in the data section it is by default an empty set. In the for–loop the constraint sets are generated for each of the local post–

conditions of the current action. In AMPL every constraint needs a name and the name has to be unique. For this reason we are composing the name from the action's name and the index of the current constraint in the ordered list of local post–conditions in line 4. The called `Constraint2AMPL` template (line 4) specifies the actual re–serialisation of the OCL constraint, which was parsed in Section 3.2.3, in AMPL syntax. As explained in Section 3.3.3 variables are represented as an array containing one value for each state during the execution. Consequently when a constraint accesses a variable we need to specify in which state that variable is accessed. That is the point where the index from the activation set is used. If the variable reference was marked with `@pre` we are accessing the variable in the previous state and thus need to subtract one from the index; otherwise we access the variable at the index from the activation set. This is shown in line 17. The serialisation of function calls and literals is straightforward as shown in line 11 and line 14. The function toOpSymbol() called in line 11 will generate the appropriate operation symbol representing the TCGOCLOperationType literal it is called on.

3.3.5 Transforming Guards

Algorithm 6 Model–to–text transformation from TCGControlFlow to AMPL model

```
1 [template public ControlFlowToAMPL(cf : TCGControlFlow)]
2   [if cf.guard != null /]
3     set [cf.name/] within {0..pathlength} default {};
4     s.t. [cf.name/]_guard {i in cf.name} : [ConstraintToAMPL(cf.
        guard)/] ;
5   [/if]
6 [/template]
```

The transformation of guards works analogously to the transformation of local post–conditions. The transformation is specified in Algorithm 6. A control flow can either have a single guard or not. Multiple guards are not possible. If there is no guard no code needs to be produced at all for this control flow. The name of a control flow is guaranteed to be unique within the activity test case graph model to avoid name clashes. In line 3 the activation set for the control flow is declared and in line 4 the indexed constraint set for the guard is declared. For the actual re-serialisation of the textual OCL in AMPL syntax the ConstraintToAMPL template from Algorithm 5 is used. Guards cannot contain variable references marked with `@pre`, thus the optional decrement of the index in line 17 in Algorithm 5 will never be added for a guard.

3.3.6 Specifying Control Flow Paths in the AMPL Data

To clarify how the control flow path will be symbolically executed and encoded in the AMPL data we will refer back to the introductory example shown in Figure 3.6. The corresponding AMPL model produced by the transformation specified in this chapter we show in Algorithm 7. The last three lines of Algorithm 7 contain the `DATA` section. In the `DATA` section we see that the `pathlength` is 1 and in the first state the constraints of the `then_Action` are active. In the initial state the constraint of the `_then` control flow is active. Since there are no cycles in the path every activation set has at most one entry. Keep in mind that the semantics of an activity test case graph is like a Petri–net. In cyclic activity diagrams there are paths that contain one control flow several times. The

Algorithm 7 Example AMPL model with corresponding data section specifying a control flow path

```
1  param Pathlength;
2  var y{0..Pathlength} : integer >=-10000, <= 10000 := 1;
3  var x{0..Pathlength} : integer >=-10000, <= 10000 := 1;
4  set else_Action within {0..Pathlength} default {};
5  s.t. else_Action_post0{i in else_Action} : (y[i])=((x[i-1])
       -(100.0));
6  s.t. else_Action_post1{i in else_Action} : (x[i])=(x[i-1]);
7  set then_Action within {0..Pathlength} default {};
8  s.t. then_Action_post0{i in then_Action} : (y[i])=(x[i-1]);
9  s.t. then_Action_post1{i in then_Action} : (x[i])=(x[i-1]);
10 set _then within {0..Pathlength} default {};
11 s.t. then_guard{i in then} : (x[i])<=(5.0);
12 set _else within {0..Pathlength} default {};
13 s.t. else_guard{i in else} : (x[i])>=(6.0);
14
15 DATA
16 param Pathlength := 1;
17 set then_Action:= 1;
18 set then:= 0;
```

activation set will then contain one entry for each state in which a token moves along the control flow from the source node to the target node. Actions can be executed several times to move from one state to another, thus the corresponding activation set will then contain the index of the state after execution of the action for each execution. An algorithm for transforming a control flow path represented as ordered set of control flows into a function mapping every instance of an AbstractTCGElement to an activation set is given in Algorithm 8. Every non–empty activation set belonging to an action having local post–conditions or a control flow having a guard needs to be expressed in AMPL syntax; the MOFM2T template for this is shown in Algorithm 9.

3.4 Abstract Test Case Generation

In order to generate a test suite with good model coverage we need to select a set of control flow paths from the activity test case graph that we want to generate test data, and finally create a unit test for. We use a depth first search or alternatively a breadth first search to find suitable control flow paths to generate unit tests from. To ensure that the search is terminating we introduced two parameterised termination conditions. The user can specify the maximum amount of test cases to be found and the maximum length of control flow paths to be tested. There are only finitely many control flow paths up to a certain length within a test case graph. We also specify early infeasible path recognition for the breadth first search. We solve the AMPL program for a control flow path during search and bound the search if the AMPL program was reported to be infeasible.
The rest of this section is organized as follows: In Section 3.4.1 we introduce basic concepts and data structures. The algorithms presented in Section 3.4.2 will use those concepts and data structures. The early infeasible path recognition is explained in Section 3.4.3.

Algorithm 8 Transformation from a control flow path to a map of activation sets

Require: path := ordered set of TCGControlFlows forming a control flow path
Ensure: activationSet contains activation sets representing the given path

```
1: activationSet(element : AbstractTCGElement) : Integer Set    ▷ Mapping from an
   activity test case graph element to a set of integers. Initially all sets are empty.
2: i ← 0
3: for all cf : TCGControlFlow ∈ path do
4:     if cf.guard. != null then
5:         activationSet(cf).add(i)
6:     end if
7:     if cf.target.oclIsKindOf(TCGAction) then
8:         i ← i+1
9:         activationSet(cf.target).add(i)
10:    end if
11: end for
```

Algorithm 9 Text template for printing an activation set in AMPL syntax

```
1 [template public ActivationSet2AMPLData{element
      AbstractTCGElement}/]
2 [if ActivationSet(element) != {}/]
3 set [element.name] :=[for integer i : ActivationSet(element)] [
      toString(i)/] [/for] ;
4 [/template]
```

3.4.1 Path Tree

For the algorithms presented in the following section we need some definitions. A control flow path is represented as a sequence of control flows. An *abstract test case* is a control flow path starting at the initial node of the activity and ending at a final node. Any node with no outgoing control flow is a final node. For two subsequent control flows, a and b, in a control flow path the OCL expression `a.target = b.source` is true.
We will need the concept of a *path tree* to explain the presented algorithms. We define the *path tree node* as a triple of an edge, the depth, and the predecessor. The edge is a reference to a TCGControlFlow within the examined activity test case graph. The depth is a natural number and predecessor is a reference to another path tree node. The depth of a path tree node is always one more than the depth of its predecessor. A root node does not have a predecessor and its depth is 0. In the described algorithms the target of a currently examined TCGControlFlow is called *current node*. We call each outgoing edge of the current node a *consequent edge*.

3.4.2 Path Search

We find all abstract test cases in the activity test case graph that have not more than maximum path length control flows. Therefore we construct a path tree from the activity test case graph. The search can be terminated when the maximum amount of test cases is reached. We have two possible orders to construct the path tree: breadth first and depth first.

Algorithm 10 Breadth first search algorithm to generate abstract test cases from an activity test case graph

Require: atcg := A TCGActivity
Require: MaxDepth := maximum path length of abstract test cases found
Require: MaxNoPaths := maximum amount of test cases to be found
Ensure: abstractTestCases := set of at most MaxNoPaths abstract test cases consisting of at most MaxDepth TCGControlFlows each

1: abstractTestCases $\leftarrow$ initially empty set of control flow paths $\triangleright$ initialisation
2: queue $\leftarrow$ initially empty queue of path tree nodes
3: **for** edge : TCGControlFlow $\in$ atcg.initialNode.outgoing **do**
4:     queu.put((edge, 0, null)) $\triangleright$ Add root path tree nodes
5: **end for** $\triangleright$ end initialisation
6: **while** queue is not empty **and** abstractTestCases->size() $\leq$ MaxNoPaths **do**
7:     ptNode $\leftarrow$ queue.get() $\triangleright$ visit next search tree node from the queue
8:     **if** ptNode.depth $\leq$ MaxDepth **then**
9:         **for** edge : TCGControlFlow $\in$ currentEdge.edge.target.outgoing **do**
10:             queue.put((edge,ptNode.depth,ptNode)) $\triangleright$ Add successor nodes to queue
11:         **end for**
12:     **end if**
13:     **if** currentEdge.edge.target.outgoing->size()=0 **then** $\triangleright$ final node found
14:         path $\leftarrow$ reconstruct control flow path from the edge in the root path tree node to ptNode.edge by traversing predecessors
15:         abstractTestCases.add(path) $\triangleright$
16:     **end if**
17: **end while**

Breadth First Search

In breadth first search we find all control flow paths of length n before a control flow path of length $n + 1$ is examined. We use a queue to store path tree nodes and examine them in first–in–first–out order. Algorithm 10 shows the pseudo code of the algorithm based on breadth first search. We initially start by generating root path tree nodes for each outgoing edge of the initial node. In each round one path tree node is retrieved from the queue. We add a new path tree node to the queue for each consequent edge if the current depth is smaller than the maximum path length (MaxDepth). When a final node with no more outgoing edges has been found, we can traverse the path of predecessors back to the root path tree node and construct an abstract test case. The algorithm will halt when either all paths up to MaxDepth length have been found or the number of paths found exceeds the maximum amount of test cases (MaxNoPaths).

Depth First Search

During the depth first search, we will always find the abstract test case next that has the longest common sub–path with the last found abstract test case. It works exactly like the breadth first search, just replace the first–in–first–out queue with a last–in–first–out stack in Algorithm 10. For the implementation this algorithm can be more efficient since it is not necessary to keep the complete path tree until the search is done. As soon as all successors of a path tree node have been found, the path tree node can be discarded.

3.4.3 Early Infeasible Path Elimination

When searching for abstract test cases as shown in the subsections before, it may happen that the constraints along a control flow path found are unsatisfiable. That means there is no valid input data that will cause the execution of this path. Let us, for example, consider a control flow path containing one TCGControlFlow with the guard $x \leq 5$ and another TCGControlFlow with the guard $x \geq 10$, and in between those two control flows there is no TCGAction changing the value of x. Obviously, there is no possible value for x fulfilling those two constraints. When no valid input data for a certain control flow path can be found we call that control flow path an *infeasible path*.

Advantages of Early Infeasible Path Recognition

The most important advantage of infeasible path recognition is that we can bound the search for abstract test cases. If a control flow path is infeasible then no extension of that control flow path will be feasible. The number of control flow paths found may grow exponentially with the maximum path length. If we assume a large graph where, in average, each node has two outgoing edges we will get 1024 paths of length 10. When one control flow path of length 3 was already infeasible then there are 128 infeasible paths of length 10. We can avoid many fruitless searches by excluding infeasible paths as early as possible. When the abstract test case generating algorithm is capable of infeasible path recognition the user can also control that the set of abstract test cases found is finite by adding additional control flows before each loop in the activity diagram. The guard of the additionally added control flow must contain a constraint on the loop variant. Thus most control flow paths containing arbitrary many iterations of this loop will be excluded from the abstract test case generation. For example, when we model a simple counter loop decreasing some counter variable i to 0 we can specify in the guard before the loop that the counter vari-

able may be between 3 and 5. In this case the path search algorithm with infeasible path recognition will find only those paths where the loop is iterated between three and five times.
Furthermore, it is useless to generate abstract test cases that are infeasible since infeasible abstract test cases can not be transformed into test cases.

The Algorithm

When using the breadth first or the depth first search approach we can generate abstract test cases and pass them on to the next step in the unit test generation algorithm, where infeasible paths will be detected. When we want to check whether a currently examined control flow sub path is a feasible path we will interleave the Abstract Test Case Generation step with the Specific Test Data Generation step.
Only a slight modification of Algorithm 10 is necessary to enable early infeasible path elimination for our breadth first search based algorithm: the condition in line 8 needs to be extended. It controls whether a path tree node will be added for the consequent edges or not. We additionally need to check whether the control flow path, which is represented by the current path tree node (ptNode), is feasible. The check for feasibility is done by generating the AMPL data for the control flow path represented by the current path tree node and solving the problem as explained in Section 3.5.2. The generation of the AMPL data from a control flow path was explained in Section 3.3.6. For infeasible path detection we will analyse the indication returned from the solver. If the solver reports 'solved', 'solved?', or 'unbound' the examined control flow path is feasible. If the solver reports 'infeasible' or 'failure' we assume the examined control flow path to be infeasible.

3.5 Specific Test Data Generation

For every abstract test case found during Abstract Test Case Generation we want to generate a test case. A test case contains input values for every parameter of the function under test, initial values for fields, and the corresponding expected output values. Test cases are stored in a unit test model. From a unit test model we can easily generate unit tests for any programming language. In Section 3.6 we will present a MOFM2T template that generates C++ unit tests from a unit test model. The unit test meta model is explained in Section 3.5.1.
In order to get numerical values for the parameters and fields of the unit under test we have a constraint satisfaction problem solved for each abstract test case. The constraint satisfaction problem to solve for each abstract test case is encoded as AMPL program according to the specification given in Section 3.3. The AMPL program will be solved by one of the solvers interfacing the AMPL system. An overview of the solvers we tried out for this thesis is given in Section 3.5.4. How we interact with the AMPL system, is explained in more detail in Section 3.5.2. When testing, often errors are triggered by test data that is exactly on the boundary of one or more decisions. For example, when we test the decision `if x<=5 then...` the test value $x = 5$ will cause the test to fail when the $\leq$ has been replaced by a $<$; other test values will not trigger this fault. How we can ensure to find such boundary values, is explained in Section 3.5.6.
The AMPL encoding ensures that we can use the same AMPL model for every control flow path within one activity. Each control flow path and any sub–path within an activity can be specified via the AMPL data, which is not part of the AMPL model and can be modified interactively. The fact that we use the same model for several solver invocations

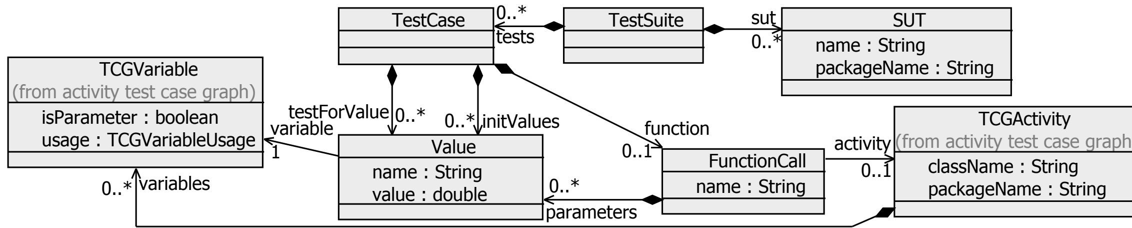

Figure 3.7: The unit test meta model

in a row enables us to use the warm start capabilities of many solvers to speed up the computation, as we will show in Section 3.5.5.

3.5.1 The Unit Test Meta Model

The unit test model serves as interface between the Specific Test Data Generation and the Unit Test Synthesis. All information needed to build a compilable unit test for the function that is modelled by the activity is contained in a unit test model. The unit test model is completely independent from the programming language and the unit test framework used for testing.
In Figure 3.7 we depict the meta model of the unit test model. The root element of a unit test model is a TestSuite. A TestSuite can contain several TestCases. Each TestCase consists of two sets of Values and a FunctionCall. The Values in the initValues reference are meant to be used for initialisation of fields. The Values referenced by the testForValue reference specify which value the corresponding fields need to hold after the tested function has been executed. If any field in the system under test does not hold the specified value the test case has failed. The FunctionCall held by the function reference has a name and a reference to the activity test case graph specifying the behaviour of the function under test. The FunctionCall also contains Values in the parameters reference. The Values referenced by the FunctionCall's parameters reference holds the input arguments to pass to the function under test.

3.5.2 Solving a Constraint Satisfaction Problem with AMPL

For each abstract test case a constraint satisfaction problem needs to be solved in order to obtain numerical values that can be used as test data in a unit test. The constraint satisfaction problem is encoded in AMPL and consists of the AMPL model that is constant for an activity test case graph and the AMPL data that is generated for each abstract test case.
The solving of constraint satisfaction problems is done by a solver hooked to AMPL. We start an AMPL instance in a separate process and communicate with this process through its standard input and output streams. First, we load the AMPL model corresponding to the activity test case graph into the AMPL console. We will also ask the user which solver to use for all control flow paths within this model and set AMPL's solver options accordingly through the AMPL console. Then we produce test data for each abstract test case.
To produce test data for one abstract test case or control flow path we reset the data currently loaded in AMPL, encode the control flow path as AMPL data according to the specification in Section 3.3.6, and load the new data in the AMPL console. Now we have

AMPL invoke the selected solver. Depending on the solver and the problem the loaded constraint satisfaction problem is an instance of, the computation time varies and we can get one of five possible indications from the solver:

solved The solver was able to compute a valid solution to the problem.

solved? The solver heuristically computed an answer but can not prove that it is correct or optimal.

infeasible There is no valid input data for the given control flow path or the used heuristics was not lucky.

unbound The given objective function has no upper bound.

failure There has been an error condition in the solver routine, for example, division by zero.

If the solver indicates ‘solved’ or ‘solved?’ we will use the values stored in the AMPL variables to produce test cases (Section 3.5.3). For all other indications we can not produce a test case from the computed data.

3.5.3 Storing the Variable Values in a Unit Test Case Model

Algorithm 11 Algorithm transforming a solution of a CSP into a TestCase

Require: testsuite := A TestSuite element used as root element to contain the produced test cases
Require: A solver has successfully solved a constraint satisfaction problem
Ensure: testsuite contains a TestCase for the last successfully solved abstract test case

1: testCase $\leftarrow$ a new TestCase instance
2: functionCall $\leftarrow$ a new FunctionCall instance
3: testCase.function $\leftarrow$ functionCall
4: **for** tcgvar : TCGBasicVariable $\in$ tcgActivity->variables **do**
5:     **if** tcgvar.isParameter = false **then**
6:         v $\leftarrow$ createValue(tcgvar)
7:         v.value $\leftarrow$ AMPL(tcgvar.name)[0]
8:         testCase.initValues.add(v)
9:         v $\leftarrow$ createValue(tcgvar)
10:         v.value $\leftarrow$ AMPL(tcgvar.name)[$last$]
11:         testCase.testForValue.add(v)
12:     **else**
13:         v $\leftarrow$ createValue(tcgvar)
14:         v.value $\leftarrow$ AMPL(tcgvar.name)
15:         **if** tcgvar.usage = in parameter **then**
16:             functionCall.parameters.add(v)
17:         **end if**
18:         **if** tcgvar.usage = out parameter **or** tcgvar.usage = return parameter **then**
19:             testCase.testForValue.add(v)
20:         **end if**
21:     **end if**
22: **end for**

When a solver successfully computed an answer for a given constraint satisfaction problem we read the assigned values and store them in a test case model. In Algorithm 11 we see the pseudo code of the algorithm creating a TestCase object from the values computed by the constraint solver. We assume that the function AMPL(‘x’) returns the content of the

Solver	LP	MILP	COP	OP	SAT	SMT	FCSP	MINLP
Cplex	✓	✓						
IlogCP[27]	✓	✓			✓	✓	✓	
GeCoDE[28]					✓	✓	✓	
JaCoP					✓		✓	
Couenne[29]	✓	✓	✓	✓				✓
Gurobi	✓		✓					
LPsolve[30]	✓	✓						
Minos	✓		✓					

Table 3.1: List of solvers and problems they can solve.

AMPL variable or variable array x as array. The function createValue(var : TCGVariable) creates a new Value object with its name attribute set to var.name and its variable reference pointing to var.
The root element of a test case model is a TestSuite. We will create one TestSuite element for a TCGActivity. For each abstract test case on that TCGActivity that has been successfully solved we will create one TestCase and add it to the TestSuite. For each TCGVariable with its isParameter attribute set to false in the TCGActivity we will retrieve an array of values from the AMPL process. The first value in this array is stored in a Value element. The name of this Value element is set to the name of the TCGVariable and it is added to the initValues reference of the TestCase. In the same way, the last element from the array retrieved from AMPL will be stored in a Value element and stored in the testForValue reference. A FunctionCall with the same name as the TCGActivity will be created. For each TCGVariable with its isParameter attribute set to true a single value will be retrieved from AMPL, and, depending on the usage property of the TCGVariable, the produced Value element will be stored either in the FunctionCall's parameters or the TestCase's testForValue reference. For out parameters and return parameters the Value elements are stored in the testForValue set. For in parameters the Value element is stored in the FunctionCall's parameters.

3.5.4 Review of Available Solvers

A central point of our approach for unit test generation is the application of state–of–the–art constraint solvers to the constraint satisfaction problem that we have generated from a UML activity diagram. As we have explained in Section 2.2 there are several specializations of the constraint satisfaction problem for which algorithms or heuristics exist. Every available solver implements one or more algorithms or heuristics. Thus a solver is suitable for instances of one or more of the presented problems. Table 3.1 contains in each row the name of the solver and a check mark for each problem we have successfully tested it for.

3.5.5 Warm Start Capabilities

To speed up the computation we use the warm start capabilities of AMPL and most of its solvers. A warm start means that we are solving the same constraint satisfaction problem multiple times in a row but each time with different data. In subsequent runs the solver can reuse knowledge it has gained throughout previous runs and thus generate solutions faster. Using the warm start capabilities requires all problems being solved in the same process

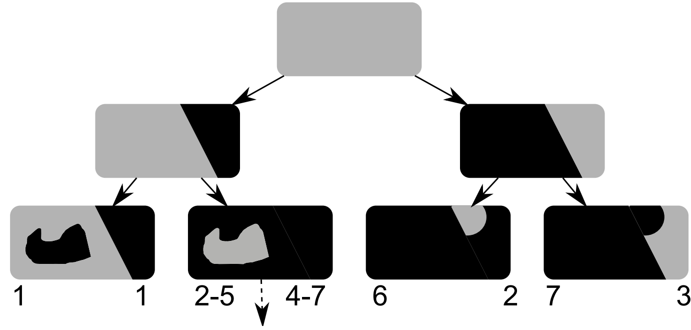

Figure 3.8: Venn diagrams for different paths and their ordering

and keeping the AMPL model constant. Keeping the AMPL model constant is achieved by the transformation of a TCGActivity to an AMPL model, which allows specifying control flow paths in the data.

Even if the solver is changed between two runs or the solver does not build internal knowledge about the given problem AMPL will store the last answer computed. Many algorithms need a starting point and they will use the last answer computed. In Figure 3.8 we show a path tree. Each leaf node represents an abstract test case; each intermediate node represents a sub–path from the initial node to some node in the activity test case graph. At each node in the path tree there is a Venn diagram visualising the feasible set of input data for that sub–path. Adding a constraint will never increase the feasible set. Consequently, the feasible set of a path extending another path will always be smaller than the feasible set of its predecessor. The feasible set of a sub–path will always be exactly the union of the feasible set of its successors and the intersection of feasible sets of siblings in the path tree will always be empty. If that were not the case there would be input data for which the control flow is unspecified or non–deterministic. The number on the left of each leaf node gives the order in which the leaf nodes are found by a depth first search and the number on the right of each leaf node gives the order in which they are found by a breadth first search.
Usually, it is easier to solve a constraint satisfaction problem if we know a starting point that fulfils all constraints except one instead of a starting point violating the majority of the constraints. If we have a solution for a sub–path it is already a feasible solution for one of its successors. If we have a solution for a leaf node we can easily obtain the solution for one of its siblings. Siblings in the path tree share all but one guard condition that means there is only one condition violated if we use the solution from a sibling as starting point. For optimisation problems we have a high chance to move into the feasible region of its sibling by applying a simple local steepest descent search on the objective function augmented by quadratic penalty terms. In the order produced by depth first search, abstract test cases sharing a long common sub–path are next to each other and will, consequently, be solved right after each other. In breadth first order we can see that the fourth abstract test case will use a starting point not fulfilling any constraint for this path. Especially when we solve the constraint satisfaction problems for the intermediate nodes in the path tree - as it is done for the early infeasible path recognition explained in Section 3.4.3 - we should memorise the solutions found and set the starting points accordingly before we solve the constraint satisfaction problem for a subsequent path tree node.

3.5.6 Boundary Value Analysis

As already motivated, a solution at the boundary of the feasible set for a certain abstract test case is more valuable than other solutions within the feasible set. For linear problems the simplex algorithm will automatically generate only boundary values since it is the nature of the simplex algorithm to move along the exposed points of a polyeder. For all other methods and generating a specific boundary value is a bit more complicated.
So far the solved constraint satisfaction problems do not contain an optimisation constraint. Thus, all algorithms terminate as soon as they found any feasible point. In AMPL we can interactively add an objective function before launching the solver for a certain problem. We add an objective function in a way that the optimal point is somewhere at the boundary of the feasible set. Any linear objective function will serve this purpose. After the addition of a linear objective function three things can happen: either the solver returns a boundary value in the next run, or the solver reports that the problem is unbounded, or the solving time for the problem suddenly increases enormously. In the first case everything went fine. If the feasible region is a convex set not containing infinity this will always be the case. The second case indicates that the direction of the objective function points into a direction, where the feasible set is open. In this case we can change the direction of the objective function and try again to hit a boundary. The last case is especially a problem of mixed integer non–linear programs and optimisation problems. It is not too hard to find a feasible solution for one of these problems, when there are enough solutions. Finding an optimal solution for a mixed integer or non–linear program is much harder. We actually don't require a global optimum; a local optimum is at the boundary of the feasible region as well. In such a case we can first solve the problem without objective, and when a feasible starting point is available, we apply a local search on the problem with objective and we will not necessarily obtain the global maximum but a local maximum that is at the boundary of the feasible set. We suggest using the last explained procedure to find as many boundary values for each abstract test case as the user demands. The used objective functions may also be provided by the user. We have successfully tested this procedure for mixed integer non–linear programs with the solvers Couenne to find a feasible point and Minos to perform the local search (Section 4.2.4). In Section 4.3.2 we will also demonstrate in our case study that generating test data at a specific boundary of the feasible region does not increase the overall runtime of our algorithm at all.

3.6 Unit Test Synthesis

From a unit test model and the corresponding activity test case graph model we are able to write compilable unit test in any language, using any unit test framework we want. In this Section we will explain an example MOFM2T template (Algorithm 12) that generates C++ unit tests suitable to test C functions. We use the Boost test library as unit test framework.
First, we need to include the Boost header and the headers for the system under test. For each TestCase in the TestSuite we print one test case. What is printed for each TestCase can be found in the lines 8–34. The central element of each test case is a call to the function under test (lines 22–23). Every time data is needed either for initialisation or to verify the results of the function call we will use a value stored inside a Value element of the unit test model. Before the function call we initialise the fields of the system under test and create local variables to pass as input arguments into the function. We also declare uninitialised local variables to receive the output parameters and the return value from the function

Algorithm 12 MOFM2T template of a C++ unit test using Boost test library

```
1  [template public UnitTestToBoostUnitTest(ts : TestSuite) { int
       count = -1; TCGVariable return = ""; }]
2    #define BOOST_TEST_MAIN MyTest
3    #include <boost/test/included/unit_test.hpp>
4    [for(sut : SUT | ts.sut)]
5      #include [sut.packageName/]/[sut.name/].h
6    [/for]
7    [for(tc : TestCase | ts.tests) { count = count + 1; TCGVariable
         ret = tc.function.activity.variables->select(v : v.usage =
         return parameter)}]
8      BOOST_AUTO_TEST_CASE( test[count/] ){
9      //field initialisation
10     [for( v : Value | tc.initValues)]
11       [v.name/] = [v.value/];
12     [/for]
13     //input parameter declaration
14     [for( v : Value | tc.function.parameters)]
15       const [v.type/] [v.name/] = [v.value/];
16     [/for]
17     //output parameter declarations
18     [for( var : TCGVariable | tc.function.activity.variables->
           select( var : TCGVariable | var.usage = out parameter))]
19       [var.type/] [var.name/];
20     [/for]
21     [ret.type/] [ret.name/];
22     //call function under test
23     [ret.name/] = [tc.function.name/]( [for( var : TCGVariable |
           tc.function.activity.variables->select( v : TCGVariable |
           v.isParameter = true and not v.usage = return parameter))
           separator(',')] &[var.name/] [/for])
24     //check fields, return parameter and output parameters for
           correct values
25     [for( v : Value | tc.testForValue)]
26       [if(v.variable.variableType=boolean or v.variable.
             variableType = integer)]
27         BOOST_CHECK_EQUAL([v.name/],[v.value/]);
28       [/if]
29       [if(v.variable.variableType=real)]
30         BOOST_WARN_EQUAL([v.name/],[v.value/]);
31         BOOST_CHECK_CLOSE([v.name/],[v.value/], 0.001);
32       [/if]
33     [/for]
34     }
35   [/for]
36 [/template]
```

call. After the function call every variable holding a value that needs to be checked will be checked. For variables in a discrete domain we check for equalities and for floating point variables we check for similarity. Thus, a test will not fail due to a small rounding error, but we will generate a warning if a small rounding error occurred (line 30).

4 Evaluation

We implemented the algorithm explained in Chapter 3 as Eclipse plug–in, successfully tested it for a set of academic problems, and performed a case study with a real world model from Airbus. We point out that the transparent interchangeability of solvers is indeed one of the biggest advantages of the approach presented in this thesis. The limitations of our approach will be discussed in Section 4.5 as well as missing but easily implementable features increasing the usability.

4.1 Experiment Description

With the test models presented in Sections 4.2 and 4.3 we performed different experiments such as measuring the overall runtime and how it depends on different parameters of the algorithms described in Sections 3.4 and 3.5. For all academic models we ran the generated test cases against an implementation and for some of them we manually performed systematic mutation tests. The details of the performed experiments are described in this section. The results will be presented respectively for each test model in Sections 4.2 and 4.3.

All experiments have been performed on a standard laptop with an Intel® core™i5 quad core processor and Microsoft® Windows 32 bit operating system.

4.1.1 Runtime Measurement

The total runtime of our algorithm is mainly influenced by two factors: the time necessary to solve one instance of the mathematical problem and the total count of problems to be solved during unit test generation. The total count of problems being solved depends on the model and the parameters used for the Abstract Test Case Generation explained in Section 3.4. For an acyclic activity diagram there are finitely many control flow paths and abstract test cases. Assuming an activity diagram with a tree–like control flow graph of depth 10 and a fanout of two in each node we get a total of 2^{10} abstract test cases. With normal breadth first or depth first path search we will have to solve one mathematical program for each abstract test case in order to obtain test data, or find out that the control flow path is infeasible. In general, we can say the number of control flow paths in an activity diagram grows exponentially with the number of decisions in the activity diagram; so does the number of problems being solved.

For the Abstract Test Case Generation we examine the influence of three parameters on the overall runtime. That is the maximum path length, the maximum number of test cases, and the number of unchecked steps. All three of those parameters directly influence the amount of problems being solved in total and therefore influence the runtime. The maximum path length bounds the path search in depth and ensures the termination of the presented algorithm. The effect of this parameter has been explained in detail in Section 3.4.2. For an activity diagram containing at least two different cycles the number of abstract test cases grows exponentially with the maximum path length. The total amount of control flow paths in a graph consisting of two alternative cycles of length a can be

calculated with equation (4.1) where l_m denotes the maximum path length.

$$\sum_{n=1}^{\lfloor l_m/a \rfloor} 2^n \tag{4.1}$$

The number of abstract test cases that need to be checked whether valid test data can be obtained for them grows linearly with the maximum number of test cases. The count of mathematical problems to be solved is the maximum number of test cases divided by the chance that any abstract test case found is an infeasible path. Consequently, the overall runtime should depend linearly on the maximum number of test cases.
The parameter for unchecked steps plays a role when the early infeasible path recognition is activated (Section 3.4.3). The early infeasible path elimination will cut off a huge amount of abstract test cases when a sub–path is found to be infeasible, but it will also slightly increase the count of problems to be solved in the worst case. We again consider the tree–like control flow graph with a fanout of two and a depth of ten. We have to solve one mathematical program for each abstract test case to obtain the test data and during the path search we will solve a mathematical program for several sub–paths. Assuming all 2^{10} abstract test cases are feasible paths the number of problems to be solved is symbolically given in equation (4.2). l_u is the number of unchecked steps.

$$2^{10} + \sum_{n=l_u+1, \Delta n=l_u+1}^{10} 2^n \tag{4.2}$$

When we determine already after the third decision one of the 2^3 control flow sub–paths to be infeasible the total number of mathematical programs being solved will reduce by $\frac{1}{8}$. The fewer unchecked steps we use, the more mathematical problems will be solved in vain if all paths are feasible, but the earlier we get the chance to eliminate infeasible paths, the larger is the fraction of the search space we can directly cut off when an infeasible path is recognised. The optimal value for the unchecked steps parameter is a trade–off between solving unnecessarily many mathematical problems and not being able to cut off large portions of infeasible abstract test cases early enough.
The runtime of a single solver run for those test models that require decidable problems with a tractable algorithm to be solved is usually below 0.1 seconds. For problems with intractable algorithms the runtime also stays below one second. A slight advantage in terms of solving speed has a large impact on the overall runtime since the solver consumes more than 95% of the overall runtime. For undecidable problems there is the problem that the heuristic might run forever and we need a good time limit for that. When the solver time limit is set too low, we will interrupt the solver many times too early although it would have found feasible test data for the current path. This will in total result in fewer test cases than possible. When the time limit is too large we can waste hours waiting for the solver and realizing that it will not return.
We measured the overall runtime of our algorithm transforming a UML *Activity* into a compileable unit test for each of the models presented in the following section. For every example model and the case study we varied some of the parameters such as the maximum path length, the maximum number of test cases, the number of unchecked steps, the used solver and its options. We plot the graphs and discuss the experimental results in Sections 4.2 and 4.3.

4.1.2 Mutation Testing

For a selection of the academic test models presented in Section 4.2 we manually performed mutation tests. Therefore we selected a set of expressions from the implementation and defined mutation operators and applied them systematically to the selected expressions. For each manually generated mutant we ran the automatically generated test suite and counted the failed tests.
A selected expression could be the predicate of a control flow statement or an assignment. As mutation operators we used the replacement of a single operation within the set of selected expressions. For example we replaced `!=` by `==` or `>=` by `<=`. In every resulting mutant exactly one operation has been replaced. Since we produced one test case for every possible control flow path in the activity diagram we expect the generated test to find most errors.

4.2 Academic Examples

In order to demonstrate the strength of our approach we built a set of test models. Each test model requires a different instance of the mathematical programming problems presented in the Mathematical Foundations to be solved in order to generate test data. Each solver is suitable for a set of problems, and for several problems there are multiple solvers that can be used. We explain of which problem the mathematical program generated for each test model is an instance and justify which solvers are suitable to solve it.
For each test model we generated test cases with the depth first search explained in Section 3.4.2 using the early infeasible path elimination explained in Section 3.4.3. With each selected solver we generated test cases for the test model using different parameters for the depth first search and early infeasible path recognition. We measured the total runtime of our algorithm and plot the results of the performed runtime measurement experiments that have been explained in detail in Section 4.1.1. For the case study we also demonstrate that it is no additional effort to generate boundary values as test data.
Finally, we built a C implementation of the function modelled by the test model and execute the generated test suite against this implementation. We also manually introduced some errors into the implementation and verified that the produced test suite contains test cases that fail on those errors and that all test pass when the implementation is correct. The manual mutation testing procedure is explained in more detail in Section 4.1.2.
The rest of this section is organized as follows: we will start Section 4.2.1 with a test model that requires instances of an easy problem to be solved in order to obtain test data. By easy we mean decidable problems with a tractable algorithm. Then we will explain in Sections 4.2.2–4.2.3 how we generated test data for problems that require NP–hard problems to be solved. Finally, we will also present in Section 4.2.4 a test model that require an instance of an undecidable problem to be solved in order to obtain test data. That means we can only generate test data with heuristic methods.

4.2.1 Triangle Classificator (LP)

We start with a model that contains only linear equations and inequalities and variables in $\mathbb{R}$. Figure 4.1 shows the activity diagram of a function taking three floating point values as input and returning a floating point value indicating whether the input variables could possibly be the side lengths of an equilateral, isosceles, or scalene triangle. The mathematical problem to solve for this model is a linear program and can be solved with the simplex method as implemented in Cplex and LPsolve or an interior point method as

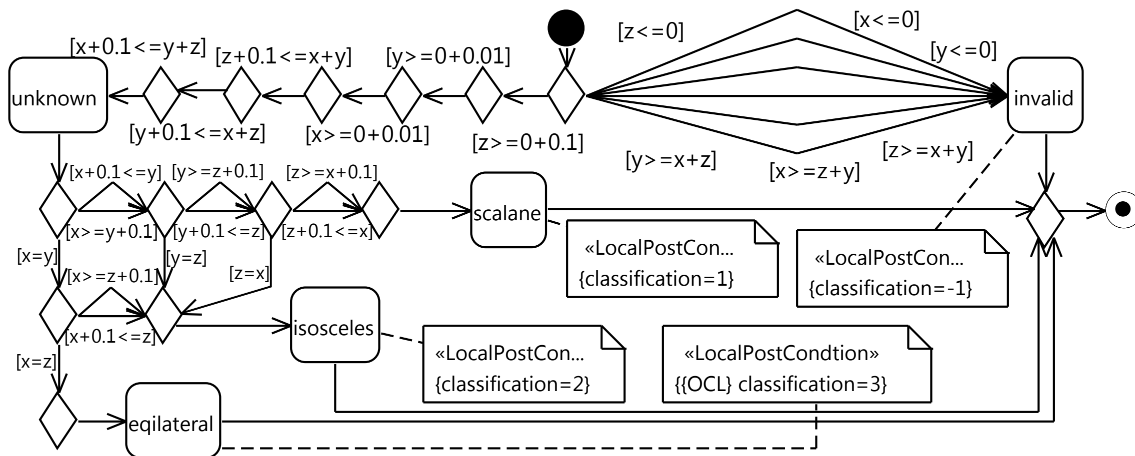

Figure 4.1: Activity diagram of a triangle classificator using only linear arithmetic constraints (LP)

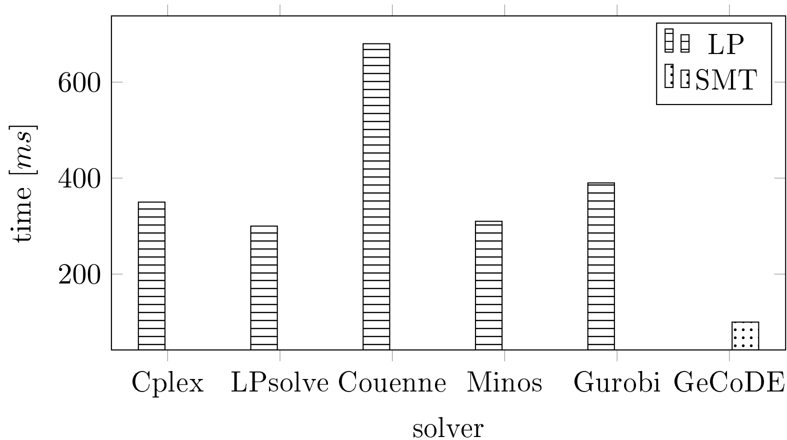

Figure 4.2: Runtime of unit test generation for the two triangle classificator models using different solvers

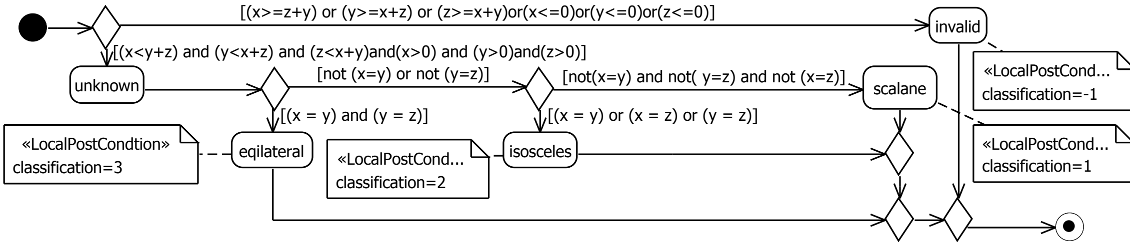

Figure 4.3: Activity diagram of a triangle classificator using logical operations (SMT)

implemented by Couenne, Minos and Gurobi.
Generating test data for every feasible abstract test case takes for every tested solver less than one second. There are in total 19 possible test cases resulting from this model. Figure 4.2 shows the average runtime consumed for test generation from this model using the mentioned solvers. Couenne is actually targeted at the much more general problem of mixed integer non–linear programming and, therefore, performs unnecessary overhead when used for solving a linear program. Minos and Gurobi implement an interior point method suitable to solve convex optimisation problems which is almost as fast as the simplex implementations of LPsolve and Cplex that are specialized for linear programming and mixed integer linear programming.
We manually performed the mutation testing as described in Section 4.1.2 for this test model. The following operator replacements have been made in the C code: `&&`→`||`, `>`→`<=`, `+`→`-`, `||`→`&&`, `==`→`!=`. This resulted in 20 mutants of which 16 have been detected by our automatically generated test cases. We have checked the remaining 4 undetected mutants all resulting from the mutation operator `&&`→`||` and found them equivalent; they therefore can not be detected. That means 100% of the detectable mutants have been detected successfully.

4.2.2 Triangle Classificator with Logical Constraints (SMT)

Since the formulation of the constraints by means of non–strict inequalities only is cumbersome we reformulated the model and used logical operations as well as strict inequalities in the guards and local post–conditions and yield the much more intuitive model depicted in Figure 4.3. The variables are now all in the domain of integers and we used logical operations as well as equalities and inequalities, consequently, this model will be transformed into an instance of a satisfiable modulo theories problem. Currently there are three solvers available for AMPL that can handle logical constraints: IlogCP, GeCoDE, and JaCoP. Since for IlogCP a license is needed we performed our experiments with GeCoDE and JaCoP. In total there are 4 possible test cases to be found and our algorithm took 110ms to generate all of them using GeCoDE. This time is also listed in the Figure 4.2. JaCoP did not succeed to solve the problem. We use +10000 and -10000 as upper and lower bounds for any variable by default. JaCoP exited indicating that too many recursive method calls have been made. After we reduced the bounds of the variables to 100 JaCoP was able to solve the problem but took on average 24 seconds. We therefore do not recommend JaCoP as constraint solver for SMT problems.

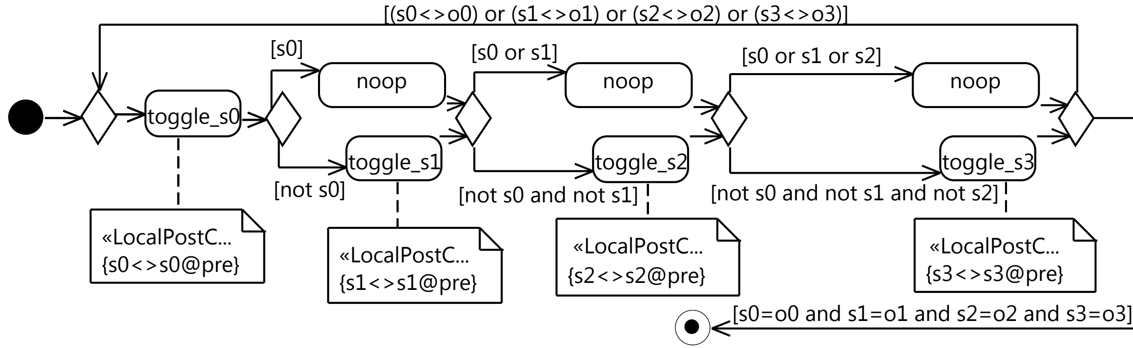

Figure 4.4: Activity diagram of a binary counter incrementing a 4 bit register

4.2.3 Binary Counter (SAT)

In Figure 4.4, we list the activity diagram of a binary counter. The variables $s0$ to $s3$ hold the current state of the counter. The activity models a function with four input arguments $o0$ to $o3$ denoting the target counter state. During execution the function will increment the binary counter stepwise until the current state of the counter is the same as the target counter state. Then the function returns. In contrast to the models presented so far this activity diagram contains a cycle and, consequently, there are infinitely many abstract test cases and we need to bound the Abstract Test Case Generation.
All variables used are in $\mathbb{B}$ and we use logical operations in the formulation of constraints. In addition to the basic logical operations we also used the equality and inequality relation for boolean variables. The equality of two boolean variables x and y can be expressed as $x \wedge y \vee \neg x \wedge \neg y$ and the inequality as $\neg x \wedge y \vee x \wedge \neg y$. We consider the problem to solve in order to find test data for this activity an instance of the Boolean Satisfiability Problem (SAT). The contained equalities and inequalities could also be considered as expressions in a theory defining equality and inequality over boolean variables. Consequently, the formulation could also be seen as SMT instance.

Experiments

We performed several experiments with this example. First, we transformed the UML activity diagram into an activity test case graph and then into an AMPL model. We verified by hand that the constraints in the AMPL model indeed express the same semantic as the OCL guard conditions and OCL local post–conditions in the UML model, and that every embedded OCL constraint contained in the test model shows up in the AMPL model. We implemented the C function that is modelled by this activity diagram in order to run the generated unit tests against it later. Then we generated test cases for the model using GeCoDE and JaCoP. We used the depth first search algorithm with early infeasible path recognition as explained in Section 3.4.2 to find test cases for this activity diagram. We generated all test cases up to a control flow path length of 80. In the right graph in Figure 4.5, we show the total number of test cases depending on the maximum path length.We also measured the overall runtime of the unit–test generation process. On the left hand side of Figure 4.5 we see the total runtime needed by our algorithm to find all test cases up to a given maximum path length using different configurations as explained in the following paragraphs. —— and - - show the runtime of of our algorithm using GeCoDE and JaCoP for the model depicted in Figure 4.4.

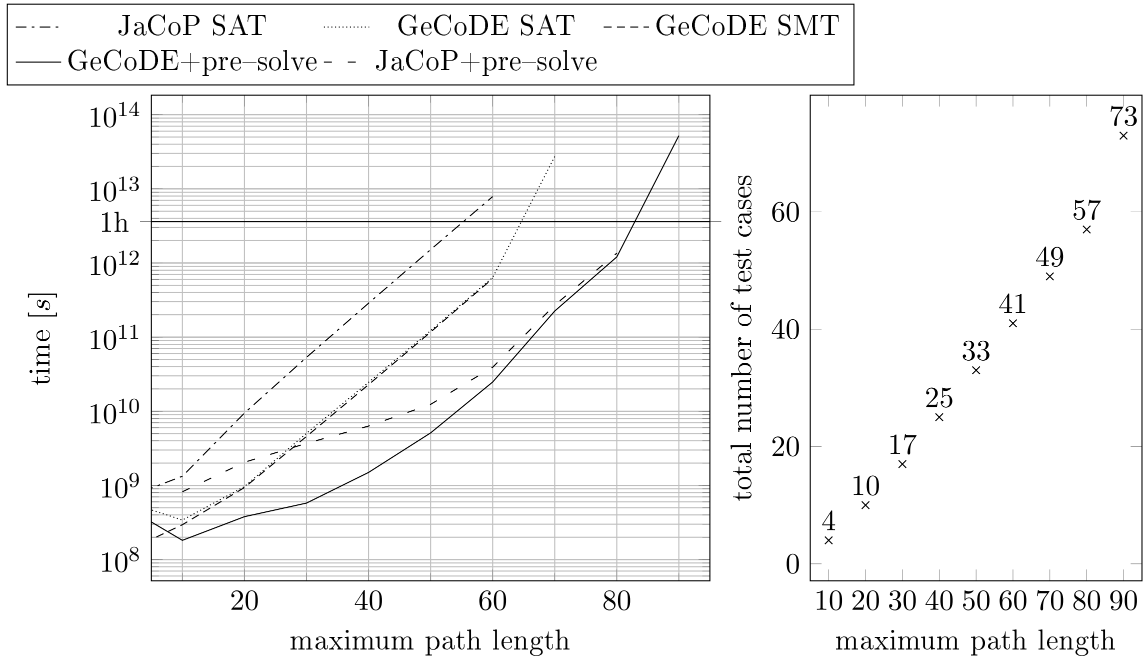

Figure 4.5: Runtime consumed by our implementation to generate all test cases for the binary counter model depending on the maximum path length. We used depth first search and different solvers as well as two alternative model formulations.

Alternative Formulation of Constraints As we have seen, the embedded OCL constraints contain equalities and inequalities over boolean variables in addition to the basic boolean operations. We replaced the equalities and inequalities with the equivalent boolean formula. Not all embedded OCL constraints were given in conjunctive normal form. Moreover the produced AMPL model will nevertheless still contain equalities, due to the continuity constraints that will be added automatically as explained in Section 3.2.4. We performed the runtime test also for the slightly modified model. The result is shown in the left graph in Figure 4.5. ········ and –·–· indicate the runtime of our algorithm using GeCoDE and JaCoP as solvers for the activity diagram modified as explained above. ---- depicts the runtime of our algorithm for the activity diagram as shown in Figure 4.4 using GeCoDE as solver. When comparing ········ and ---- we see that for the solver it does not matter how the logical constraints are formulated. Those two plots are almost identical.

Effect of AMPL's Pre–Solve Phase AMPL tries to tighten the bounds of variables, eliminate some variables, and also eliminate some constraints that have no effect on the feasible set before passing the problem to the solver. This step is called the pre–solve phase and is activated by default. For the modified version of the original model we experienced a serious memory leak in AMPL caused by AMPL's pre–solve phase. Thus we deactivated the pre–solve phase. The pre–solve phase caused AMPL to consume huge amounts of memory; and AMPL crashed when trying to allocate more than two gigabyte. By deactivating the pre–solve phase the runtime of our algorithm for the initial binary counter model using equalities increases by a factor of at least 100. This is due to the fact that most problem instances can already be identified as infeasible by the pre–solve phase after a few microseconds while solving the problem with a solver at least takes the time to start the solver in an external process. In Figure 4.5, the —— plot denotes the runtime of the algorithm using GeCoDE as solver with the pre–solve phase enabled and ---- plots the runtime with AMPL's pre–solve phase disabled. Both plots are using the

initial model. The plots –·–· and ········ show the runtime for the modified model with the pre–solve phase deactivated, since it was not possible to generate test data for the modified model otherwise. As we can see our algorithm is much faster with AMPL's pre–solve phase enabled.

Mutation Testing For this model we also performed manual mutation testing with the resulting test suite. We generated the test suite for all paths up to a maximum length of 40 using GeCoDE as solver. The test suite contains 25 test cases. The C implementation of the binary counter is straightforward consisting of a `do while` loop and the bit flipping is done in the loop body. We applied the following mutations to the predicate of the `do while` loop in our source code: `||`→`&&`, `!=`→`==`, `!=`→`<`, `!=`→`>`. This results in 15 different mutants out of which 10 have been detected by the generated test suite, 66.$\bar{6}$% of the generated mutants have been detected.

Discussion As already explained in Section 2.2.6 the SAT problem is decidable but in its general form there is no tractable algorithm for it. Still, our problem size will stay below the threshold where the solver needs more than a second to solve it. The number of solver invocations grows exponentially with the maximum path length when generating one test case per feasible control flow path through the activity diagram. In order to find the 49 possible test cases with a control flow path length of up to 70 our implementation solved 421901 SAT instances. Thus any slight advantage of one solver over the other or the way to formulate a constraint in terms of solving speed has an enormous impact on the overall runtime of the algorithm. As we saw during the experiment, replacing equalities and inequalities with boolean formulas has no impact on the runtime but disabling AMPL's pre–solve phase had a negative influence on the overall runtime of our algorithm. Furthermore, the plots in Figure 4.5 reveal a super–exponential growth of the runtime of our implementation although we actually expected only exponential growth of runtime for our algorithm.

4.2.4 Exploding Tyres (MINLP)

Figure 4.2.4 shows an activity modelling the physical process of pumping air into a tyre. The relations between volume, amount of air (n), pressure, and temperature are described by the ideal gas equation. The equations for adiabatic compression of air during one pump stroke states a non–convex relation between these physical measures. Additionally we introduced the boolean variables tyreExploded and tyreMelted, which will be set when the pressure or temperature inside the tyre raised above a certain threshold. Moreover we introduce an integer loop counter that determines how many pump strokes will be executed. This model requires mixed integer non–linear programs to be solved in order to generate test data. The solver Couenne from the COIN-OR project is suitable for this task. Couenne is the only mixed integer non–linear programming solver we tested.

Runtime Measurement

Mixed integer non–linear programs are in general undecidable. Consequently, it may happen that a solver trying to solve an instance of an MINLP runs infinitely long. It usually uses less than one minute to generate a feasible solution for one problem instance. But it also might run for a very long time or even infinitely long.
In Figure 4.7, we depict the overall runtime of the test generation for this model using

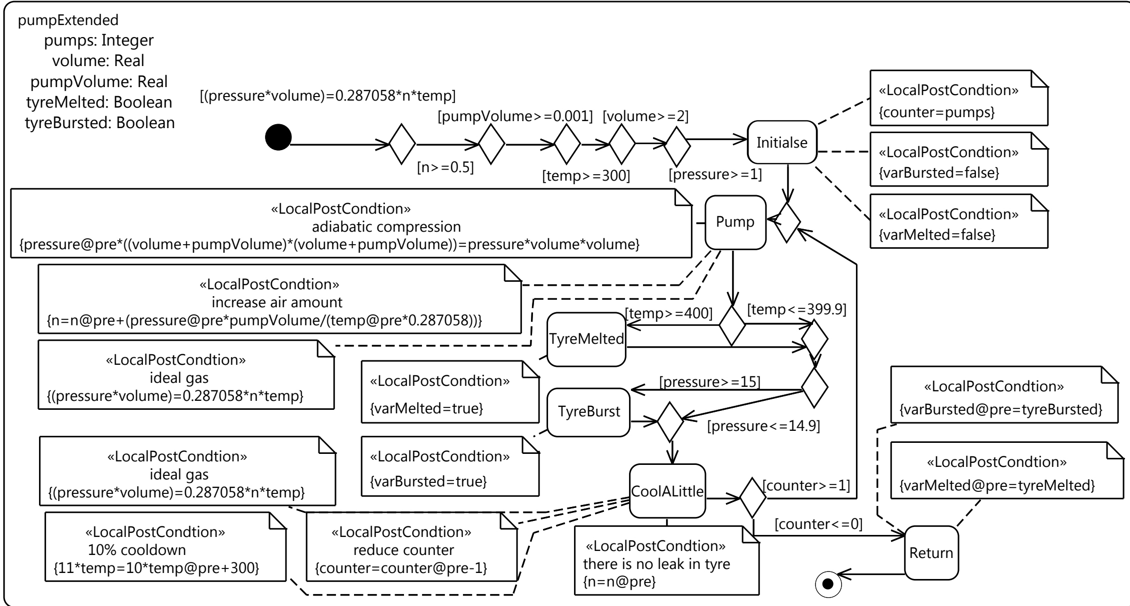

Figure 4.6: Activity diagram with mixed integer non–linear constraints

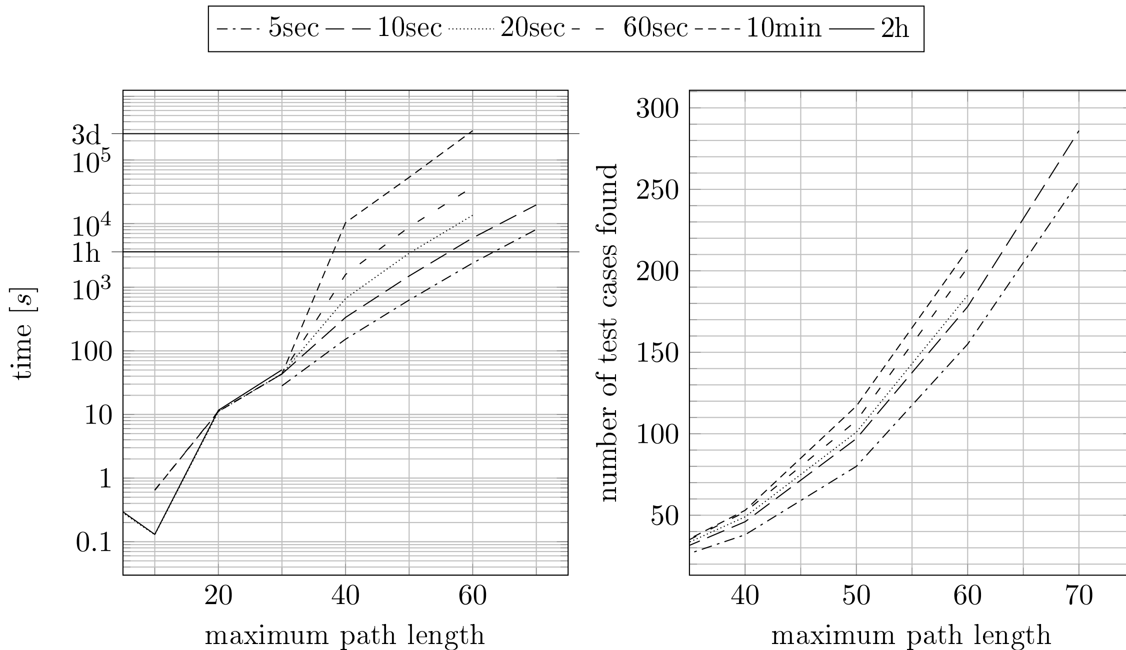

Figure 4.7: Runtime consumed and test cases found by our implementation for the tryre pump model. We used depth first search and Couenne as solver. The maximum path length and the solver time limit have been varied.

Couenne. For each plot, we used the solver time limit printed in the legend. On the right hand side of Figure 4.7 we depict the amount of test cases found depending on the time limit and the maximum path length. Up to a path length of 30 all problem instances had been solved or found infeasible after less than 20 seconds, thus the set time limits did not make a difference in the runtime as well as for the number of test cases found up to a maximum path length of 30. Comparing the run time of our algorithm with different time limits for the maximum path length of 60 we recognise that increasing the time limit from 20 seconds to 60 seconds increased the overall runtime by a factor of 2.78 and generated 17 more test cases; those are 9.2% more test cases. Increasing the time limit from 60 seconds to 600 seconds increased the overall runtime by a factor of 7.67 and produced 11 additional test cases, which amounts to only 5.4% more test cases. We see that increasing the lime limit for the solver massively increases the overall runtime, but we gain only very little more test cases for that. We therefore recommend using 10 to 20 seconds as time limit for Couenne.

Running the Generated Test Cases

We have implemented the modelled function in C and successfully ran the generated test cases against it. Although we did not do it systematically for this model we did a few mutation tests for the test cases generated from this model. The error propagation of very small errors in the floating point calculations is especially interesting. We see in the model there is a loop and in every round the values for temperature, pressure, amount of air (n in the model) are updated depending on the values of those measures before. Consequently, when one of the calculations has a small error the error will be propagated to all three of those measures and we see a lot of failed tests due to just one wrong statement. Also for longer test cases taking the loop multiple times the introduced error is propagated super–linearly. In an experiment an error of 0.1 % per loop iteration caused after 5 iterations an error of more than 0.5%. Similarly, reordering of statements introduced huge errors in all three measures. And finally, a sub–optimal calculation plans caused several test cases to fail due to rounding errors. One might think that the new temperature after the CoolALittle action could be computed either as `temp = (10 * temp + 300) / 11 ;` or as `temp = 10 * temp / 11 + 300 / 11 ;`. But the automatically generated test cases revealed that those two expressions do not produce the same results. The rounding errors of the second version cause all generated test cases to fail.

4.3 Case Study PAX Model (MILP)

Additionally to the artificial test models demonstrating the use of our method for different kinds of constraints we also performed a case study with a real world model. We tested our implementation on a model modelling the PAX call system from Airbus. Out of the model of the complete product we selected an activity diagram modelling the interaction of the PAX call system with the **I**n **F**light **E**ntertainment (IFE) system. The selected *Activity* contains 21 *Actions*, 24 *ControlNodes*, and two *LoopNodes*. Furthermore, there are eight *DataStoreNodes* representing function local variables. The model was originally created with Atego® Artisan Studio, and used to generate a C implementation from it with a proprietary code generator from Atego®. The branching conditions and the code body of each *Action* is given in C syntax. All assignments and conditions consist of linear equations and inequalities only. All variables are in the integer or boolean domain. Consequently, the constraint satisfaction problem to be solved for test data generation is a mixed integer

linear program. The solvers Cplex and LPsolve are perfectly suitable for this kind of problem. Couenne can also be used for this task.
The described algorithm has several parameters, which mainly influence the Abstract Test Case Generation. We use this case study to evaluate the influence of those parameters on the runtime of the algorithm. The examined parameters are the maximum length of control flow paths, the maximum amount of test cases to be generated, and the unchecked steps before an early infeasible path check is performed during the depth first search.

4.3.1 Manual Adaptation

In order to generate C++ unit test code with our Eclipse plug-in from the described model several pre–processing steps need to be performed manually. We need to convert the model from Atego® Artisan Studio into Eclipse UML, add all guards and the local post–conditions in OCL syntax, flatten the *LoopNodes*, and replace the *DataStoreNodes* by *Properties*.
Atego® does not store models natively in XMI format but has an option for exporting models in XMI format. The Eclipse Modelling Framework natively uses the XMI format to store models. Each modelling tool uses its own implementation of the UML meta model and there are slight differences in the implementations making the created models incompatible with each other. In order to import the model from Artisan in Eclipse we manually need to remove some objects not recognised by Eclipse and correct some typing errors in the XMI file. Then we can load the XMI file and browse the model with the Eclipse Modelling Framework.
Every *Action* is associated with a C code snippet that is used for code generation. There are not yet *Constraints* specifying the pre and post–conditions of each *Action* in OCL. Also the *guards* do not contain OCL queries. We make an educated guess to add *guards* and *localPostconditions* reproducing the semantics of the C code snippets contained in the original model. The original model used local C struct variables modelled by the *DataStoreNodes* contained by the *Activity*. Our implementation can handle primitive data type variables modelled by *Properties*, consequently we created one *Property* per field of a struct variable. The original model also used arrays. We emulated the behaviour of an indexed collection by allowing all variables depending on an index to change to an arbitrary value whenever the index is changed. This may produce wrong behaviour but seemed fair enough to evaluate our algorithm. The *DataStoreNodes* will be ignored by our algorithm.
The *LoopNodes* contain further model elements in their *bodyPart* reference. Those elements from the *bodyPart* are directly included into the parent *Activity*. The *InitialNode* and *FinalNodes* of the loop body are replaced by decision nodes and directly connected to every element that was connected to the *LoopNode*. In order to preserve the loop semantics additional *ControlFlow*s, *DecisionNode*s, and counter incrementing *Action*s are added.
The function specifying which *ControlFlow* to take after each *Action* is well–defined and defined over the complete domain. Well–defined means there is no possible state in which more than one *guard* evaluates to true, and defined over the complete domain means that there is no value assignment for which every *guard* of the *outgoing ControlFlow*s evaluate to false.

4.3.2 Runtime Measurement Results

We examine the influence of the used solver, the maximum path length, and the maximum number of test cases on the runtime of our algorithm. The expected effects of those parameters on the overall runtime have already been discussed in Section 4.1.1.

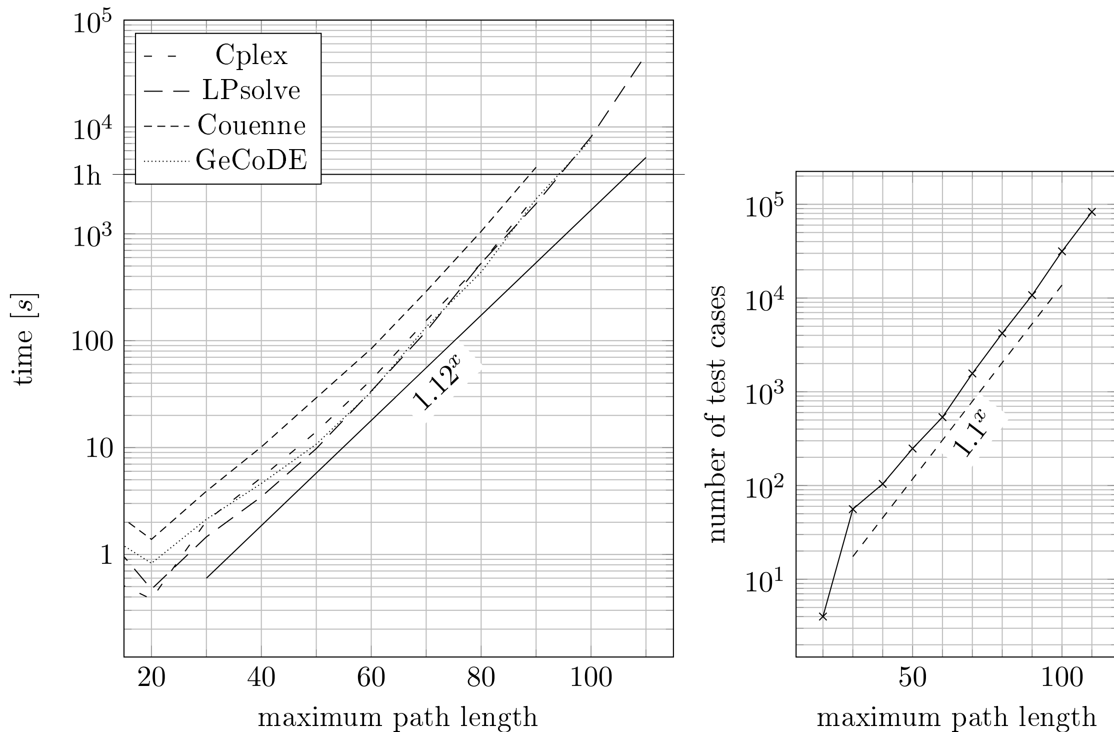

Figure 4.8: Runtime of our algorithm applied to the case study. We measured the runtime with different solvers and varied the maximum path length. The runtime as well as the amount of test cases grow exponentially with the maximum path length.

Different Solvers

We plotted the runtime depending on the maximum path length on the left hand side of Figure 4.8. We performed this experiment with different solvers. For Abstract Test Case Generation the depth first search with early infeasible path elimination and two unchecked steps has been used. As we see, LPsolve Cplex and GeCoDE are equally fast. Using Couenne and a maximum path length of 60, our algorithm takes round about three times the runtime consumed using LPsolve and twice the runtime consumed with Cplex. Keep in mind that the plot is logarithmic. This might be because Couenne is actually suitable for the much more general mathematical problem of mixed integer non–linear programming; it is not perfectly specialized for mixed integer linear programming. To evaluate the slope of the exponential growth we also plotted the function $A \cdot 1.2^x$. The runtime grows almost perfectly exponential with the maximum path length and the runtime doubles when the maximum path length increases by 6-7. With a closer look at the graph, we can see that it is slightly super–exponential. This might be caused by effects that are inherent to the computing architecture executing the implementation of our algorithm.
The case study model contains only constraints that are instances of MILP and, therefore, the solvers always succeed to find the solution when there is suitable test data for a certain path. Consequently, our algorithm will produce all test cases up to the given path length no matter which solver is used. The number of test cases grows equally exponential with the maximum path length as the runtime. We plotted the number of test cases depending on the maximum path length on the right of Figure 4.8. This plot is also logarithmically scaled.

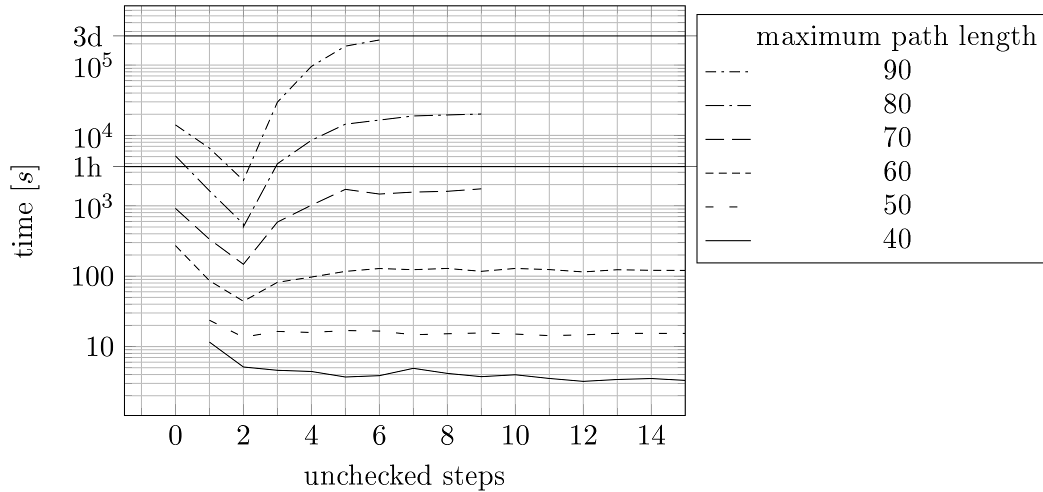

Figure 4.9: Runtime of our implementation depending on the unchecked steps. The experiment has been performed with depth first search and Cplex as solver for different maximum path lengths

Unchecked Steps

In Figure 4.9, we plotted the overall runtime of our algorithm depending on the unchecked steps of the early infeasible path elimination. We used Cplex as solver for this experiment and repeated the experiment for different maximum path lengths. In the plots we see, up to a maximum path length of 50 it is not beneficial to use early infeasible path elimination when generating test cases for the case study model. With longer maximum path lengths the impact of well configured early infeasible path elimination grows. For a maximum path length of 90 our algorithm configured with the optimal value for the unchecked steps parameter takes less than one hour to generate all test cases, while it takes several days when unchecked steps is set to 5 or 6.

Maximum Amount of Test Cases

The runtime of our algorithm is not the only thing that grows exponentially with the maximum path length. Also the number of control flow paths in the activity diagram shorter than the maximum path length grows exponentially. We show this effect in Figure 4.8. The number of test cases grows exponentially with respect to almost the same basis as the runtime and the quotient of runtime and generated test cases grows polynomial. To illustrate that, we plotted the runtime of our algorithm against the number of test cases in Figure 4.10. For the plot on the left we used the same data as for the LPsolve plot in Figure 4.8. The plot is logarithmic on both axis and we can see that the runtime grows not linearly with the number of test cases, but it grows with an exponent of 1.45.
It would not produce useful test cases to use depth first search and no limit on the maximum path length but a limit on the test cases to find. All generated test cases would over and over share very long common sub–paths and other control flows would not be checked at all. Since this would not be desired in practice, we used the breadth first search explained in Section 3.4.2 to evaluate the effect of the maximum amount of test cases on the runtime. In Figure 4.10, we plot on the right the runtime of our algorithm with breadth first search, early infeasible path elimination, and Cplex as solver. The dotted line shows a small section

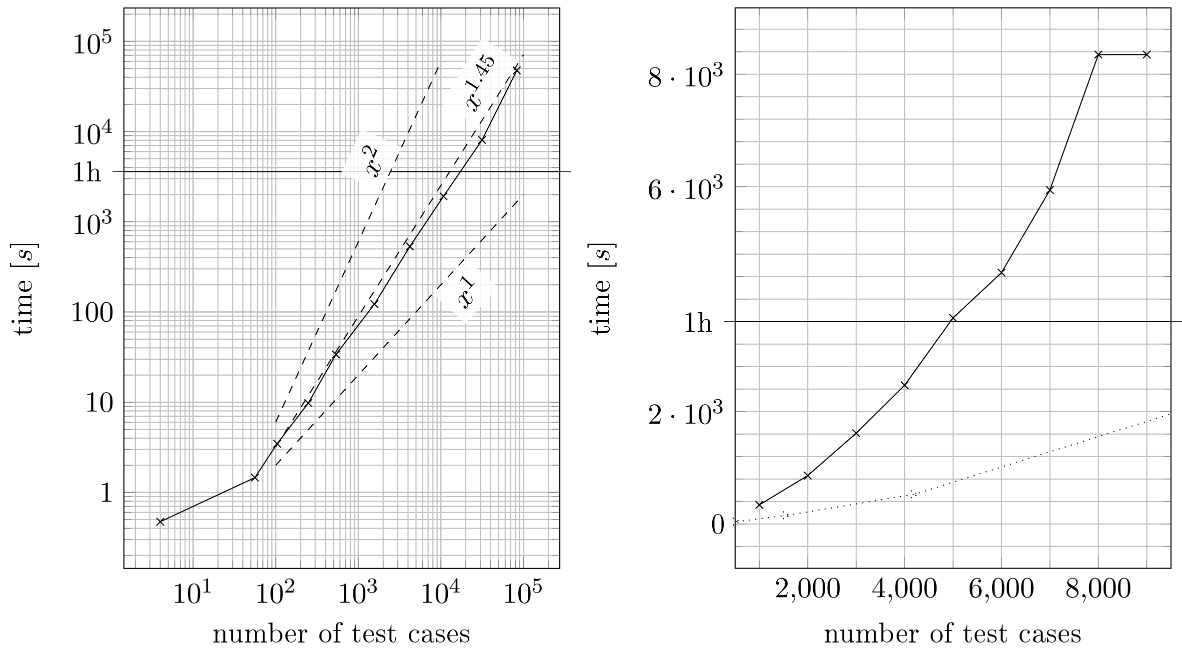

Figure 4.10: Runtime of our implementation depending on the number of test cases. Left: depth first search. Right: breadth first search

of the first plot where the previous data has been reused. As we can see breadth first search is considerably slower than depth first search. The reason for that is that breadth first search can not make such a good use of the warm start capabilities of the solvers as depth first search can. Details of the warm start capabilities have been explained in Section 3.5.5. In depth first search between two subsequent solver invocations usually only a small amount of the constraints change. For breadth first search it will happen several times during the infeasible path elimination that all constraints have changed between two subsequent calls to AMPL. Furthermore, we see that the generation of 9,000 test cases took only slightly longer than the generation of 8,000 test cases. We currently can not explain this strange effect.

Boundary Value Analysis

Finally, we also analysed the impact of the boundary value analysis on the runtime of our implementation. As explained in Section 3.5.6, an additional objective in the AMPL program has been added that enforces all initialisation values of the test case to be minimal with respect to the path constraints in the control flow path. In Figure 4.11, we show the runtime of our algorithm depending on the maximum path length. We use depth first search and LPsolve as solver. The runtime with boundary value analysis and the runtime without boundary value analysis are plotted. Obviously it does not make a difference at all for the runtime whether we generate values at a specific boundary of the feasible region or just any feasible points as test data.
This is plausible because the number of solver invocations where really a boundary value analysis is performed is small compared to the number of total solver calls. For example, for a maximum path length of 70 there are 1568 test cases. The solver performs the optimisation only for them. On the other hand, there are 35044 solver invocations in total including those for the infeasible path elimination and those that recognise an abstract test

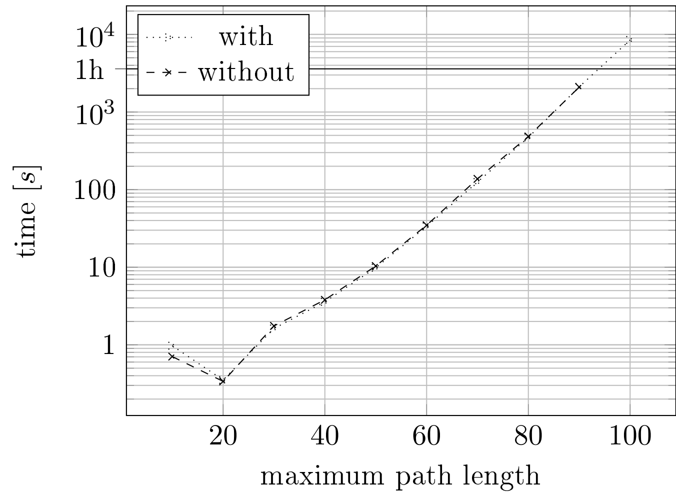

Figure 4.11: Comparison of runtime with boundary value analysis and without boundary value analysis

case as infeasible. They are performed just in the same way as without boundary value analysis. The simplex algorithm will first determine the feasibility and then optimise. Consequently, only $\frac{1}{22}$ of all solver invocations is actually performing some extra work. For larger maximum path lengths this fraction is declining. For this particular instance of the mixed–integer linear programming problem it is particularly easy for the simplex algorithm to find the boundaries since there are no interdependencies between the variables. Each variable in the objective function can just be set to its minimum and no branch–and–bound steps need to be performed.
We definitely recommend to use this feature for mixed integer linear programs in practice because it really comes at no cost, and one can use this feature to generate test data at two or more different boundaries of the feasible set and generate multiple test cases for one control flow path that check all path conditions for the control flow path.

4.4 Verification of the Implementation

We claim that the Eclipse plug–in built aside this thesis implements the algorithm explained in Generating Unit Tests from a UML Activity. This claim is validated manually by checking each step separately. We use the academic example models presented in Section 4.2 as test cases to verify each transformation described in Generating Unit Tests from a UML Activity. From each test model we created manually the intermediate artefacts that should have been created according to the specification and compared them to the intermediate artefacts produced by the plug-in.

4.5 Limitations

We demonstrated that our algorithm is capable of generating ready–to–use C unit test from the test models presented in this chapter. There are still more features to implement and also limitations that are not easily fixed. We divided this section into three parts. First, we will show which features will provably be never realizable due to theoretical constraints.

Those are the presented in Theoretical Limitations. Then, we will name features that could have easily been added to the presented method but are not described in this thesis. And finally, we will show which features could be added if the used libraries and external tools would provide better support in Limitations of Tool Support.

4.5.1 Theoretical Limitations

The most important source of limitations is undecidability. Although we were able to produce some valid test cases for a model using undecidable formulas in their local post–conditions and guards there will never be an algorithm that is able to automatically produce one test case for every possible abstract test case in the activity test case graph generated from such an activity diagram. There will always be some special cases where any applied heuristic will neither find a feasible solution nor can generate a proof that there is no solution to the mathematical program to be solved in order to generate the test data.
If all constraints in the model are instances of a decidable problem it may still be the case that the algorithm is intractable. In such a case we will have problems with the scalability of our approach when using exact solvers. The only way out would then be to use a solver implementing a tractable heuristic instead of an exact method to solve the mathematical program, which in turn might report some paths to be infeasible although there exists a solution that would have been found by the exact method.
Another limitation to scalability is the combinatorial explosion of possible control flow paths through an activity test case graph with growing path length. Unless one decides to use only a special subset of all possible control flow paths as abstract test cases the algorithm will always suffer from exponential runtime and it will also produce a number of test cases that grows exponentially with the maximum path length.

4.5.2 Limitations of the Implementation

Several features of the UML could have easily been supported by our approach. We concentrated on the core idea in this thesis and created a good software architecture allowing later extensions of our initial concept.
Missing features are support of hierarchical modelling with *ActivityCallActions* or *StructuredActivityNodes*. Furthermore, the input language currently does not support operations on variables with a type other than boolean, integer, or real. Support for arbitrary datatypes such as enums or objects can be added by modifying the Rigorous Mathematical Programming step of our algorithm.
Currently as constraints only the *ControlFlow*'s *guard* conditions and the *Action*'s *localPostConditions* are considered. All possible ways to embed invariants and pre– and post–conditions into an UML model are ignored. This could be fixed in the Normalisation step of the presented algorithm.
Finally, it might not be in the interest of the industrial user that for larger models the test case generation with our algorithm may take several days and produce several thousand test cases (for the case study we had produced 83,000 test cases in 13 hours). The generation of a moderate amount of test cases ensuring full code coverage of the implementation or fulfilling some coverage criterion on the input model might be more desirable. To achieve this a modification of the Abstract Test Case Generation is necessary.

4.5.3 Limitations of Tool Support

AMPL's normal use case is mathematical optimisation and throughout the years several additional features have been added to the language. Unfortunately we do not know about a single solver capable of handling every constraint that can be expressed with AMPL. Especially support for SMT formulations using convex optimisation and mixed integer non–linear programming as background theory is missing. With the commercial solver IlogCP it is possible to delegate the solution of sentences in a background theory to Cplex, which is capable of solving mixed integer linear programs and mixed integer quadratic programs, yet another specialization of mixed integer non–linear programming.
The used Eclipse OCL parser by default supports a minimal set of arithmetic operations. Consequently, it is not possible to support arbitrary arithmetic without modifying the Eclipse OCL parser.
In theory, the XMI interchange format should enable the exchange of UML models between different modelling tools. Unfortunately, the import of UML models into Eclipse exported by Atego® Artisan Studio does not yet work flawlessly. Consequently, our implementation is restricted to models generated by Papyrus, a modelling plug–in for Eclipse, unless one is willing to spend some manual work to make the exported UML model of third party tools compliant to Eclipse UML.

5 Outlook and Summary

5.1 Outlook

The presented algorithm and its implementation have potential for improvement. Our initial concept for boundary value analysis can be extended and become a full grown support for data based coverage criteria. Therefore we need to implement an algorithm that automatically generates objective functions from the model. Depending on the desired data based coverage criterion for each path a suitable set of objective functions can be generated and used to find all test data fulfilling the data–based coverage criterion.
In order to make an industrial–strength tool out of this prototypical implementation, one should improve the scalability of the Abstract Test Case Generation step. Currently we support an infeasible model structure–based coverage criterion. When the total number of test cases to generate depends linearly on the number of edges or nodes in the activity test case graph, the overall runtime of our algorithm will be reduced to depend polynomially on the number of nodes or edges in the activity test case graph. For example one can ensure that for each edge there is at least one test case traversing this edge. Or one can use the test goal management implemented in ParTeG. This allows selecting multiple coverage criteria that shall be fulfilled by the generated test suite and steers the test case generation accordingly [31].
Another potential is buried in the mathematical modelling system. The modelling capabilities of AMPL are currently not used to their full extent. AMPL natively supports arrays with a fixed length. Our current algorithm interprets every *Property* or *Parameter* with a basic data type as a single variable. We could also interpret the multiplicity of *Parameters* or *Properties* and interpret them as arrays of fixed lengths. This could be done with almost no implementation effort.
Besides arrays AMPL also natively supports more mathematical operations than those that are predefined in basic OCL and via an external library support for a huge selection of scientific arithmetic functions is available. By extending the OCL parser, functions like `sin`, `exp`, and different probability distributions can easily be made available to the formulation of constraints.
Another improvement which is also provisioned is the handling of object type variables. This would need some modifications in the way how activity test case graphs are transformed into AMPL models and also requires an extension of the OCL subset that is accepted as input language for constraints.
Finally, the transformation from UML into an activity test case graph could be extended such that more modelling elements from the UML are supported. Support for *ActivityCallActions* or *LoopNodes* can be added by inlining the referenced activity of an *ActivityCallAction* or the body of a *StructuredActivityNode*. We can also support arbitrary *DataTypes* by allowing a user–defined mapping of data types to one of the basic data types: boolean, integer, or real. For example, the user could use the *DataTypes* named ‘uint8’, ‘uint16’, and ‘uint32’; one can define in the model that they are integers with a specified value range.

5.2 Summary

We presented an approach for automated generation of unit test from UML activity and class models. The presented approach consists of five steps transforming an UML activity diagram into a unit test through four intermediate artefacts. We also specified at least one algorithm for each step and demonstrated the practicality of our approach by implementing it as Eclipse plug-in and applying it to a set of academic test models and a model of a real industrial software implementation from Airbus. With mutation testing we demonstrated that the presented approach is useful and fully working. We examined in detail the influence of different parameters of our algorithm on the total runtime and performed a parameter tweaking.
We presented an efficient way to integrate state–of–the–art constraint solvers and provide support for a wide variety of mathematical problems. Depending on the specific needs of the modeller we can allow a wide variety of constraint expressions to be used in the test model. All constraint solvers are accessed via a common interface, thus, making it easy to exchange the constraint solver and selecting the one that is most suitable to generate test data for one specific activity diagram. We presented several example models, each holding constraints that are instances of a different mathematical problem. We argued which solver is suitable for which model and also measured the runtime of the implementation. We clearly saw that the automatic test data generation is especially fast when constraints are specified in terms of a decidable theory with a tractable algorithm, for example, linear programming. On the other hand we also successfully demonstrated test data generation for models whose constraints are formulated as instances of an undecidable problem, for example, mixed integer non–linear programming. Although it is in general possible with our approach to generate test data for models with undecidable constraints it can be much more time consuming than generating test data for models with decidable constraints and it is not guaranteed that an existing solution will be found. We recommend restricting oneself to linear inequalities or another decidable constraint formulation, if that is possible. One should use mixed integer non–linear constraints or SMT formulas with an undecidable theory only for small models and if it is unavoidable.
We have presented a concept for early infeasible path elimination during generation of abstract test cases and examined its impact on the overall runtime of our algorithm. We have also examined the influence of several parameters on the runtime of our implementation. Using early infeasible path elimination with two unchecked steps massively reduces the runtime of our algorithm especially for larger models when one test case is generated for every feasible path with a maximum path length of 60 control flows or more. Of course, those results strongly depend on the activity diagrams used for this examination, but we assume that our case study model is a good representative of actual models occurring in practice.
Finally, we argued that with the presented approach it is possible to steer the generation of test data in a way that boundary values are produced with no additional effort. It is common knowledge that the use of boundary values as test data tends to trigger existing bugs with a higher probability. Thus we improved the quality of the generated test cases with a minimal additional effort.

Bibliography

[1] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[2] H. Lackner, H. Schlingloff, and A. Berlin, "Modeling for Automated Test Generation – A Comparison," in *Model–Based Development of Embedded Systems (MBEES)*, vol. 8, 2012, pp. 57–70. [Online]. Available: http://www.vldb.informatik.hu-berlin.de/~hs/Publikationen/2012_MBEES_Lackner-Schlingloff_Modeling-for-Automated-Test-Generation--A-Comparison.pdf

[3] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *«UML»' 99 – The Unified Modeling Language*, ser. Lecture Notes in Computer Science, R. France and B. Rumpe, Eds. Berlin, Heidelberg: Springer, 1999, vol. 1723, pp. 416–429. [Online]. Available: http://dx.doi.org/10.1007/3-540-46852-8_30

[4] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," in *«UML»' 2000 – The Unified Modeling Language*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, vol. 1939, pp. 383–395. [Online]. Available: http://dx.doi.org/10.1007/3-540-40011-7_28

[5] S. Weißleder and D. Sokenou, "Automatic Test Case Generation from UML Models and OCL Expressions," in *Software Engineering (Workshops)*, 2008, pp. 423–426.

[6] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating Test Cases from UML Activity Diagram based on Gray–Box Method," in *Asia–Pacific Software Engineering Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 284–291. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/APSEC.2004.55

[7] D. Kundu and D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams," *Journal of Object Technology*, vol. 8, no. 3, pp. 65–83, 2009. [Online]. Available: http://dx.doi.org/10.5381/jot.2009.8.3.a1

[8] A. D. Brucker and B. Wolff, "On theorem prover-based testing," *Formal Aspects of Computing*, vol. 25, no. 5, pp. 683–721, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00165-012-0222-y

[9] P. Patel and N. Patil, "Test case formation using UML activity diagram," *World Journal of Science and Technology*, vol. 2, no. 3, pp. 57–62, 2012. [Online]. Available: http://worldjournalofscience.com/index.php/wjst/article/view/13160

[10] S. Weißleder, "Test Models and Coverage Criteria for Automatic Model–Based Test Generation with UML State Machines," Ph.D. dissertation, Humboldt University Berlin, 2010. [Online]. Available: http://model-based-testing.de/data/weissleder_phd_thesis.pdf

[11] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic Test Case Generation for UML Activity Diagrams," in *International Workshop on Automation of Software Test*. ACM, 2006, pp. 2–8. [Online]. Available: http://dx.doi.org/10.1145/1138929.1138931

[12] J. Peleska, E. Vorobev, and F. Lapschies, "Automated Test Case Generation with SMT–Solving and Abstract Interpretation," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi,

Eds. Berlin, Heidelberg: Springer, 2011, vol. 6617, pp. 298–312. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20398-5_22

[13] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," in *International Conference on Quality Software (QSIC)*, 2011, pp. 41–50. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2011.17

[14] M. P. Krieger and A. Knapp, "Executing underspecified OCL operation contracts with a SAT solver," *Electronic Communications of the EASST*, vol. 15, 2008. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/view/176

[15] J. Malburg and G. Fraser, "Combining Search–based and Constraint–based Testing," in *International Conference on Automated Software Engineering.* IEEE Computer Society, 2011, pp. 436–439. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100092

[16] N. Tillmann and J. Halleux, "Pex-White Box Test Generation for .NET," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds. Berlin, Heidelberg: Springer, 2008, vol. 4966, pp. 134–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79124-9_10

[17] A. Eiben and Z. Ruttkay, "Constraint Satisfaction Problems," in *Handbook of Evolutionary Computation*, D. F. T. Bäck and M. Michalewicz, Eds. IOP Publishing Ltd. and Oxford University Press, 1997, pp. 1–8.

[18] S. Boyd and L. Vandenberghe, *Convex Optimization.* New York, NY, USA: Cambridge University Press, 2004.

[19] G. Dantzig, *Linear Programming and Extensions.* Princeton University Press, 1963.

[20] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem–Proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, jul 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[21] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer, 2011, vol. 6806, pp. 171–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_14

[22] C. Bessiere, "Constraint Propagation," CNRS/University of Montpellier, Tech. Rep., 2006.

[23] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[24] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Duxbury Press, 2002. [Online]. Available: http://www.ampl.com/BOOK/index.html

[25] *OMG Unified Modeling Language™ (OMG UML), Superstructure*, Object Management Group Std., May 2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/

[26] *OMG Object Constraint Language (OCL)*, Object Management Group Std., May 2012. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1

[27] P. Laborie, "IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ser. Lecture Notes in Computer Science, W.-J. Hoeve and J. Hooker, Eds. Berlin, Heidelberg: Springer, 2009, vol. 5547, pp. 148–162. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01929-6_12

[28] Gecode Team. (2006) Gecode: Generic Constraint Development Environment.

[Online]. Available: http://www.gecode.org

[29] P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter, "Branching and Bounds Tightening Techniques for Non–Convex MINLP," *Optimization Methods and Software*, vol. 24, no. 4-5, pp. 597–634, 2009. [Online]. Available: http://dx.doi.org/10.1080/10556780903087124

[30] M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve : Open source (Mixed-Integer) Linear Programming system. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[31] S. Weißleder and D. Sokenou, "ParTeG–A Model-Based Testing Tool," *Softwaretechnik–Trends*, vol. 30, no. 2, 2010. [Online]. Available: http://pi.informatik.uni-siegen.de/stt/30_2/01_Fachgruppenberichte/TAV/TAV29P08Sokenou.pdf