Generating Test Data from a UML Activity using the AMPL Interface for Constraint Solvers

Felix Kurth[1], Sibylle Schupp[1], and Stephan Weißleder[2]

[1] Hamburg University of Technology, Institute for Software Systems,
Schwarzenbergstr. 95, 21073 Hamburg, Germany
{felix.kurth,schupp}@tu-harburg.de
http://www.sts.tu-harburg.de
[2] stephan.weissleder@gmail.com
www.model-based-testing.de/person/stephan_weissleder

Abstract. Testing is one of the most wide-spread means for quality assurance. Modelling and automated test design are two means to improve effectivity and efficiency of testing. In this paper, we present a method to generate test data from UML activity diagrams and OCL constraints by combining symbolic execution and state–of–the–art constraint solvers. Our corresponding prototype implementation is integrated in the existing test generator ParTeG and generates C++ unit tests. Our key improvement is the transparent use of multiple industry strength solvers through a common interface; this allows the user to choose between an expressive constraint language or highly optimised test data generation. We use infeasible path elimination to improve the performance of test generation and boundary value analysis to improve the quality of the generated test data. We provide an industrial case study and measure the performance of our tool using different solvers in several scenarios.

Keywords: Model–Based Testing, Activity Diagram, AMPL, Constraint Solving, Infeasible Path Elimination, Boundary Value Analysis, Mixed Integer Non–Linear Programming

1 Introduction

Model–Based Engineering is a promising technology for system engineering. The Unified Modelling Language™ (UML) is the quasi standard for model–based specifications. A UML activity diagram can be used to give a quick and intuitive overview of a system. It can also be used to formally describe details of a procedural implementation. They can be refined in a step-wise manner starting with a vague description of the intended system usage and adding more details later on.

Modelling is not an end in itself. The benefits of Model–Based Engineering are automated subsequent tasks in software development like, for example, testing. We want to automatically generate test data from an activity diagram with embedded constraints described using the Object Constraint Language (OCL).

The test model describes the relevant control flows in the system under test (SUT). The embedded OCL constraints describe interdependencies between the variables and how variables change their value.

In this paper we present a transformation from an activity diagram into an 'A Mathematical Programming Language' (AMPL) program. For that, we symbolically execute control flow paths in the activity diagram and encode each of them as AMPL program. The solutions of these resulting AMPL programs contain input values and corresponding oracle values that can be used to test the implementation. Since test data at the boundaries of path constraints has a higher probability to detect failures, as shown in [5], we also integrate boundary value analysis in test generation. Depth–first search is used to find control flow paths and we introduced *early infeasible path elimination* within the depth–first search to reduce the runtime of our algorithm. The presented transformation is implemented as part of a master thesis [7] in a proof of concept fully automated unit test generation tool called Activity Tester (AcT). The tool is integrated as a feature into the Partition Test Generator (ParTeG) available as Eclipse plug–in [3].

The paper is structured as follows. Section 2 contains the related work. An introduction to AMPL and the used models is given in Section 3. Section 4 contains a description of the core ideas of the proposed algorithm. The report on our case study is given in Section 5. The paper is concluded in Section 6.

2 Related Work

There are several approaches for test generation based on activity diagrams. For instance, Wang Linzhang et al. propose in [9] a path–search–based method to find test scenarios in an activity diagram. They implemented the proof–of–concept tool UMLTGF. Minsong et al. [11] propose to randomly generate Java unit tests and match the test execution traces to the control flow paths in the activity diagram. They then select a subset from the test cases that covers all simple paths in the test model. Our method is also path–search–based like [9] and unlike [11] we are using constraint solvers to find test data.

Weißleder and Sokenou presented a data–oriented approach to select test cases [16]. They use abstract interpretation to derive partitions and corresponding boundaries of the domain of input values. Weißleder [15] describes a detailed collection of different model–based structure- and data–oriented coverage criteria including their formal definition. The source code of ParTeG, the proof–of–concept tool associated with this thesis, is freely available [4]. 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. In contrast to the data–oriented test data generation with abstract interpretation we are using symbolic execution. We are also offering boundary value analysis based on mathematical optimisation instead of data–partitions.

[3] ParTeG and AcT are available from: http://parteg.sourceforge.net/
[4] Source code of ParTeG is available from: http://sourceforge.net/p/parteg/code

There are several approaches to integrate the solution of OCL formulas. Ali et al. [1] use evolutionary algorithms to search for potential solutions of OCL constraints. Krieger and Knapp [6] use a SAT solver to find instantiations of variables satisfying OCL constraints. They transform OCL formulas into boolean formulas and use a SAT-solver-based model finder for solving these formulas. In contrast, we generate models for OCL formulas not based on a single heuristic or a single solver, but we are using the AMPL frontend to which we can link either a heuristic or an algorithm.

Malburg and Fraser [10] 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 [14] from Microsoft® research is based on dynamic symbolic execution. Both [10] and [14] execute the implementation with random input values and generate new input values by collecting path conditions along the executed control flow. Then they negate one of the path conditions and use a constraint solver to find a solution. Like the presented work, we also use symbolic execution. Since we are only performing a static analysis, however, we do not need the source code to generate test cases. So in contrast to the related work, our tool is also suitable for black–box testing.

3 Preliminaries

In this section, we introduce the programming language AMPL and the semantics of the used test models.

3.1 AMPL and its Solvers

We use 'A Mathematical Programming Language' (AMPL) by Robert Fourer [4] to formalise the execution of a control flow path in the test model. In this section, we describe AMPL and introduce the solvers we have used to generate test data [5]. In AMPL we can state linear, non–linear, and logical constraints. Variables can be continuous or discrete. The AMPL system serves as a common

Solver	MILP	NLP	SMT	MINLP
Cplex	✓			
LPsolve[3]	✓			
Gurobi		✓		
Minos		✓		

Solver	MILP	NLP	SMT	MINLP
Couenne[2]	✓	✓		✓
IlogCP[8]	✓		✓	
GeCoDE			✓	

Table 1. List of solvers and problems they can solve.

[5] A bundle containing AMPL and the solvers Cplex, LPsolve, Gurobi, and Minos is available from: http://ampl.com/try-ampl/download-a-demo-version/

interface to a variety of solvers. Depending on the constraint satisfaction problem encoded in the AMPL program a suitable solver can be selected to solve the problem. If all constraints are linear we use a solver that is specialized on mixed integer linear programming (MILP), for example, Cplex or LPsolve. If constraints are non–linear but convex and all variables are continuous we use a non–linear programming (NLP) solver. For problems with non–linear constraints and discrete as well as continuous variables we need a solver capable of solving mixed integer non–linear programming (MINLP), for example, Couenne [2] [6]. If logical operations are used in the constraints, a satisfiability modulo theories (SMT) solver supporting appropriate background theories is used, for example, GeCoDE [7] or IlogCP [8] [8]. In Table 1 we show a selection of solvers interfacing with AMPL. A checkmark in a cell means that we have successfully tested the solver in the row on the problem named in the column.

3.2 Semantics of the Test Models

For the transformation into 'A Mathematical Programming Language' (AMPL) presented in Section 4.2 we clarify the semantics of the expected input model. Modelling elements of the UML such as *Activity* and UML references *name* or *ownedParameter* are typeset in a special font. In this section, we first detail which UML modelling elements we use and will then shortly recapitulate the Petri–Net semantics of activity diagrams (see [12]). Further, we introduce the notion of a *state*, which will be relevant for our algorithm.

As test model we assume an activity diagram with *Action*s, *ControlNode*s, and *ControlFlow*s modelling the control flow of an *Operation*. The *Activity* is linked as *method* to its specifying *Operation*. Each *Action* can contain several textual OCL constraints as *localPostcondition* and each *ControlFlow* can hold a textual OCL constraint as *guard*. The textual OCL in the *guard* and *localPostcondition* will be parsed in the context of the specifying *Operation*. That means the OCL constraints can access all *ownedAttribute*s of the *Class* containing the specifying *Operation*. Further, all *ownedParameter*s of the specifying *Operation* can be referenced in the textual OCL. We interpret every *Property* as variable that can change its value during the execution of an *Action*. A *Parameter* will be interpreted as a parameter that can not change its value during the execution of an *Action*.

When executing a control flow path, we start with a token in the *InitialNode*. We allow only one *InitialNode* per *Activity*. The token can move along an enabled *ControlFlow*. A *ControlFlow* is enabled when the OCL constraint in its *guard* evaluates to true. We say that an *Action* is being executed when a token resides in the *Action*.

We refer to an assignment of all variables and parameters as *state*. While the token resides in the *InitialNode* there is an initial state. After each execution of an

[6] Couenne is available from: http://www.coin-or.org/download/binary/Couenne/

[7] GeCoDE is available from: http://www.gecode.org/

[8] IlogCP and its AMPL drivers are available on request from: http://ampl.com/try-ampl/get-a-trial-license/

Action the current state can change according to the OCL constraints contained in the *Action*'s *localPostcondition*. An OCL constraint contained as *localPostcondition* can specify a relation between the current state and the previous state. Consequently, the states are interconnected with each other via *localPostcondition*s. The set of all relations contained in *localPostcondition*s can be seen as state transition function. For a *ControlFlow* to be enabled the OCL constraint in its *guard* has to evaluate to true with respect to the current state. OCL constraints contained in a *guard* can only specify a relation between the variables and parameters within a single state. It is not possible to access the value of a variable in the previous state within a *guard*.

4 The Algorithm

In this section we explain the core ideas of the algorithm that we implemented in AcT. First, we demonstrate with a small example how the execution of a control flow path can be formalised. Then we show how to acquire test data that is at the boundary of path constraints and, finally, we present an algorithm that efficiently searches for executable control flow paths.

4.1 Overview

Symbolic execution is done by transforming an activity diagram into a parameterised AMPL model representing all relevant OCL Constraints, *Properties* , and *Parameter*s. This transformation preserves the original semantics of the activity diagram.

The parameters of the AMPL model encode a control flow path. An assignment of each variable in each state is generated by state–of–the–art constraint solvers from an AMPL model with given parameters. The generated variable assignment is suitable test data. A great advantage of a commonly used mathematical programming language is that industrial–strength solvers are available for a wide variety of problems (see Section 3.1).

4.2 AMPL Transformation

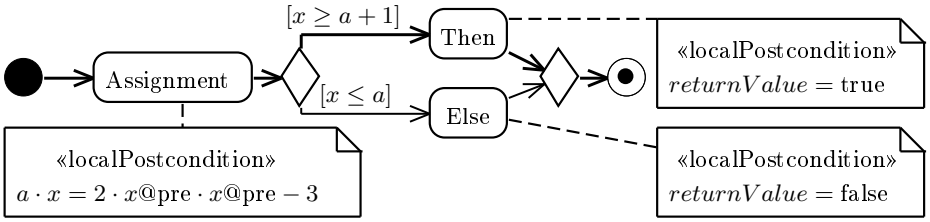

Fig. 1. A simple UML activity diagram

An AMPL model encodes the relevant *Properties*, *Parameter*s, and OCL constraints contained in an activity diagram. *Properties* and *Parameter*s are relevant if they are referenced in a *guard* or *localPostcondition* inside the *Activity* under consideration. We call a sequence of *ControlFlow*s *control flow path*. For two subsequent *ControlFlow*s A and B in a control flow path it holds that A.target=B.source. A control flow path is encoded in the AMPL data.

We model the execution of a control flow path as a series of states. Each AMPL model has one parameter called `pathlength` representing the number of *Action*s on a control flow path. Executing an *Action* changes the state, consequently, there are `pathlength + 1` states. Since a *Property* can have a distinct value in each state, they are modelled as an array of variables. *Parameter*s are constant and are therefore represented by a single AMPL variable. Every *Property* and *Parameter* has either integer, float, or boolean as *type*; in AMPL we specify the domain of an AMPL variable correspondingly.

The OCL constraints in Figure 1 reference three different OCL variables. We assume a and $returnValue$ to be *Parameter*s that are modelled as a single variable in AMPL and we assume x to be a *Property* that is modelled as an array of variables. We further assume that all three UML elements have the *type* integer. Consequently, we generate the following AMPL code for them:

```
var a : integer;
var returnValue : integer;
var x{0..pathlength} : integer;
```

During execution, a *ControlFlow*'s *guard* or an *Action*'s *localPostcondition* is always evaluated with respect to the current state. The set of states in which an *Action* or *ControlFlow* is being executed is called *activation set*. There is one activation set for each *ControlFlow* having an OCL *guard* and each *Action* having an OCL *localPostcondition*. All OCL *localPostcondition*s and OCL *guard*s are contained in the AMPL model as constraints, which can be switched on and off for each state. In the AMPL model states are referenced via their index. For each *Action* and *ControlFlow* the activation set is declared as a subset of the set `{0..pathlength}`. Each OCL constraint in a *guard* or *localPostcondition* is transformed into an indexed collection of constraints over the activation set.

When a variable reference is 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. The AMPL model code generated for the Assignment *Action* in Figure 1 is, for example, as follows:

```
set Assignment within {0..pathlength} default {};
s.t. Assignment_post0 {i in Assignment} : a*x[i]=2*x[i
    -1]*x[i-1]-3;
```

The code generated for the Then and Else *Action*s, as well as, *ControlFlow*s with *guard* is analogous. The complete AMPL model consists of the declaration of the parameter `pathlength`, variable declarations and the constraints together with their activation sets. The value of the parameter `pathlength` as well as the elements of each activation set will be specified in the AMPL data. 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.

Let us, for example, encode the bold printed control flow path from Figure 1 as AMPL data. Starting at the *InitialNode*, the current state is the initial state with the index 0. In this state there are no constraints. When we reach the Assignment *Action* our current state switches to 1. Since there is a post–condition for Assignment, we add 1 to the activation set of the Assignment. The next *ControlFlow* has an OCL guard, consequently we add 1 to its activation set. Further, we add 2 to the activation set of the Then *Action*. In AMPL syntax, the data specifying this path is:

```
param pathlength := 2;
set Assignment := 1;
set Cf_Then := 1;
set Then := 2;
```

4.3 Boundary Value Analysis

According to [5] test data at the boundaries of the path constraints is more valuable than arbitrary test data causing the given control flow path to be executed. We will refer to test data at the boundaries of path constraints also as *boundary test data*. Up to now we did not explicitly require the test data to be boundary test data. An additional linear objective function in the AMPL model ensures that generated test data is boundary test data. When solving AMPL programs augmented by such an objective function three things can happen: either the solver generates boundary test data, or the solver reports that the problem is unbounded, or the solving time for the problem increases enormously and the solver exceeds its time limit. The first case is the good case. In the second 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 programming solvers. It is not too hard to find a solution to a mixed integer non–linear program, if there are enough solutions, but finding a solution that is optimal with respect to an objective function is much harder. In fact Couenne will try to find the global optimum although also a locally optimal solution—with respect to a linear objective function—would be suitable as boundary test data. In such a case we can first use a global mixed integer non–linear programming solver to solve the original AMPL program. In a second step we add the linear objective function and use a local search algorithm using the solution generated before as starting point to find a solution at the boundary of the path constraints.

4.4 Early Infeasible Path Elimination

Control flow paths are stored in a *path tree*. A path tree is a tree data structure consisting of path–tree nodes. Each path–tree node holds a reference to its parent path–tree node, a reference to a *ControlFlow*, and an integer—its depth. The root

node points to an *outgoing ControlFlow* of the *InitialNode*. Any path–tree node is a representation of a control flow path from the *InitialNode* to the *target* of its referenced *ControlFlow*. Leaf path–tree nodes reference a *ControlFlow* that ends in an *ActivityNode* with no *outgoing ControlFlow*. We call the control flow path represented by a leaf path–tree node an *abstract test case*.

Not every control flow path is a *feasible path*. A control flow path is feasible, if there exists test data that causes this control flow path to be executed. In other words, for an infeasible path the corresponding AMPL program will contain contradicting constraints, for example, $x_0 \leq 5; x_0 \geq 10$. It is quite obvious that if a path–tree node represents an infeasible path any of its children nodes will also represent an infeasible path. Since we are only interested in abstract test cases for which test data can be generated, we prune branches containing no feasible abstract test cases as early as possible.

Pseudo code constructing a path tree with early infeasible path elimination is given in Algorithm 1. Infeasible path check (isFeasiblePath()) is done by running a solver on the AMPL program corresponding to the examined control flow path. If the solver reports a failure or infeasibility of a problem, we assume the control flow path to be infeasible. The earlier during construction of the path tree a control flow path is detected to be infeasible the more control flow paths are pruned. On the other hand, if we check feasiblility for every single path–tree node we will impose unnecessary work. We allow for a customised trade–off by introducing the parameter *unchecked steps* (UchkSteps). Further, our path search algorithm accepts the parameters *maximum path length* (MaxPathLen) bounding the depth of the path tree and *maximum number of test cases* (MaxNoPaths) bounding the number of abstract test cases to find. The function countPathDecisionsTo() in the third last line of Algorithm1 counts the number of path–tree nodes with at least 2 children nodes on the path from its source to its argument.

5 Results

The presented algorithm is implemented as an Eclipse plug–in. The tool is called Activity Tester (AcT) and is integrated into the Partition Test Generator (ParTeG). We built a set of example models with corresponding C implementations to test our implementation. Finally, our tool has also been applied in an industrial case study at Airbus Operations GmbH. The constraint solver based approach presented here has already been adopted by Amin Rezaee for *StateMachine*s [13]. In this section we will first evaluate AcT for a model with mixed integer non–linear constraints and then summarise the results of the industrial case study.

5.1 Mixed Integer Non–Linear Example Model

Figure 2 shows an activity diagram 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

Algorithm 1 path search algorithm with early infeasible path elimination

Require: root : path–tree node ▷ the root path–tree node
MaxPathLen : integer ▷ maximum path length for each control flow path
MaxNoPaths : integer ▷ maximum amount of leaf path–tree nodes to find
UchkSteps : integer ▷ number of path decisions without infeasible path check
Ensure: constructs path tree starting from root using early infeasible path elimination
decisions : integer ← 0 ▷ count path decisions
stack : LIFO Buffer ← {root} ▷ only root node in stack
while !stack→isEmpty() **and** path tree contains less than MaxNoPaths leaf nodes **do**
ptn : path–tree node ← stack→pop() ▷ remove first path–tree node from stack
f : integer ← ptn.controlFlow.target.outgoing→size() ▷ fanout of current node
if f>0 **and** ptn.depth<MaxPathLen **and** (decisions<UchkSteps **or** f<2 **or** ptn→isFeasiblePath()) **then** ▷ evaluate isFeasiblePath() only if its value is relevant
for cf : *ControlFlow* ∈ ptn.controlFlow.target.outgoing **do**
stack→push(**new** path–tree node(ptn, cf, ptn.depth+1))
end for
decisions ← (f≥2 ? (decisions mod UchkSteps)+1 : decisions)
else
decisions ← max(decisions − stack→top()→countPathDecisionsTo(ptn), 0)
end if
end while

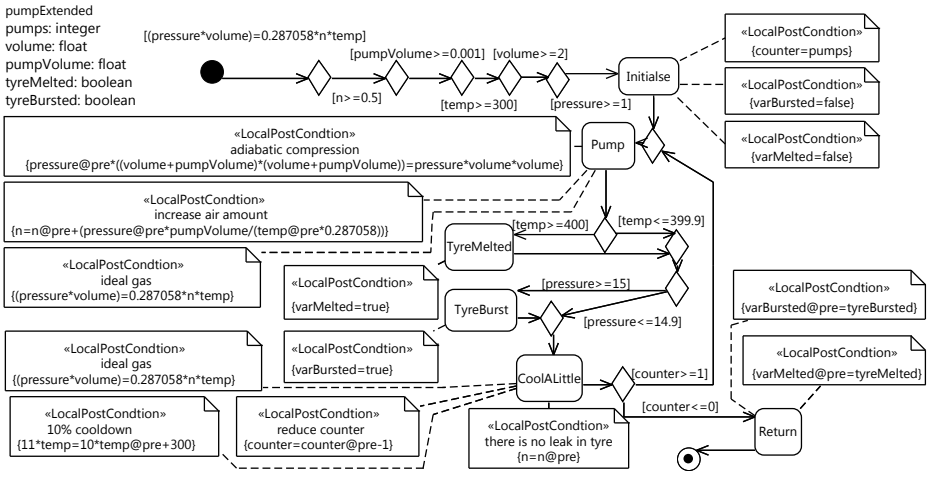

Fig. 2. Activity diagram with mixed integer non–linear constraints

compression of air during one pump stroke states a non–convex relation between these physical measures, consequently the resulting AMPL programs will be non–convex. 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 (*counter*) to count the number of pump strokes. Consequently, the AMPL program is also a mixed integer problem.

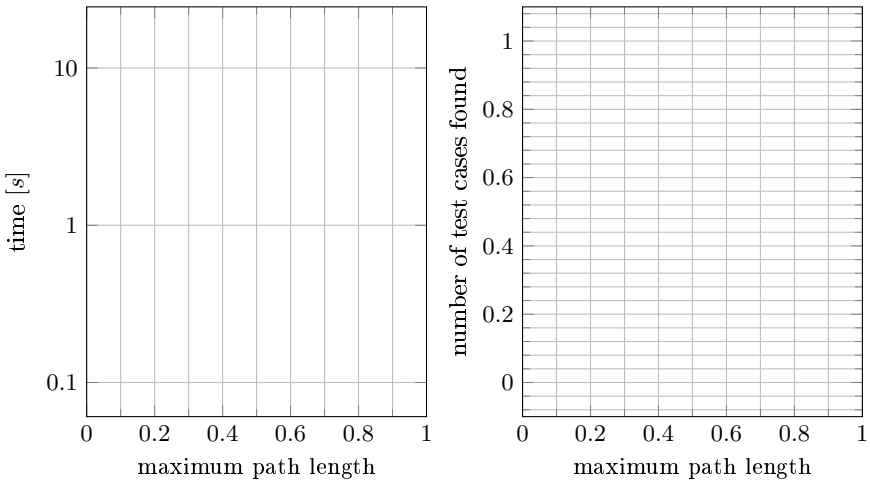

Fig. 3. Runtime consumed and test cases found for the tyre pump model

Runtime Measurement. Mixed integer non–linear programming is in general undecidable. Consequently, solvers might not halt on an MINLP instance. During test generation we solve lots of MINLP instances. It is advisable to set a suitable solver time limit. For very small time limits test generation is faster while for longer time limits chances to find a solution for a particular MINLP instance are rising. In other words, when setting a larger time limit we will certainly find a solution more often and thus generate more test cases but the overall runtime will grow. Note that this is only the case if constraints are stated by means of an undecidable theory; otherwise we can always find test data for a feasible path.

We generate test cases for the *Activity* from Figure 2. Figure 3 plots the overall runtime of the test generation depending on the maximum path length. We used Couenne as solver and varied the time limit per MINLP instance between 5 seconds and 10 minutes. The right hand side of Figure 3 plots the number of abstract test cases for which we found test data with the given solver time limit. The number of abstract test cases grows exponentially with the maximum path

length and so does the overall computation time and the number of real test cases.

Comparing the runtime of our algorithm with different time limits for the maximum path length 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 real 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 time limit for the solver massively increases the overall runtime, but we gain only very few additional test cases for that and we doubt that those additional test cases massively increase the quality of the test suite. We therefore recommend using less than 20 seconds as solver time limit for Couenne.

Mutation Testing We applied selective mutation generate 1000 mutants of our C implementation of the tyre pump model. The mutate.py [9] script used does this by text pattern matching and therefore created 427 non compilable mutants. We generated test suites with a solver time limit of 10 seconds and for a maximum path length of 20,30,40, and 50 and ran them on the 573 compilable mutants. The test suite for path length 50 was mutation adequate. The test suites for path length 20 left 20 mutants alive; that is a mutation score of 96.5%. Table 2 summarises the results of the performed mutation testing.

maximum path length	20	30	40	50
number of test cases	4	18	47	97
killed mutants	553	569	570	573
alive mutants	20	4	3	0
mutation score	96.5%	99.3%	99.5%	100%

Table 2. Mutation test results for tyre pump model

Those high mutation scores have two reasons. First, we produced one test case for every control flow path up to a given maximum path length which is quite exhaustive testing for a small academic example. Second, due to the inter-dependency between *temperature*, *pressure*, amount of air (n) a small error in one of these measures will propagate to all three of those measures causing a great number of failed tests due to just one wrong statement. Similarly, reordering of statements introduced huge errors in all three measures. And finally, sub–optimal calculation plans resulting in small rounding errors have been detected by almost every test case. For example the C expressions `(10 *temp + 300) / 11` and `10 * temp / 11 + 300 / 11` have been identified as not equivalent. Even the replacement of $<$ by $\leq$ has been detected.

[9] mutate.py is available from: http://archive.today/wz4TF

5.2 Case Study

We tested our implementation on a model from Airbus Operations GmbH. Out of the model of the complete product we selected a large activity diagram modelling the control flow of a function. The selected *Activity* contains 21 *Actions*, 24 *ControlNodes*, and two *LoopNodes*. Furthermore, there are eight *DataStoreNodes* representing function local variables. 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. All variables are in the integer or boolean domain. Consequently, for test data generation mixed integer linear programs have to be solved. Cplex and LPsolve are perfectly optimised for this kind of problem while Couenne is suitable for a more general class of problems.

The algorithm described in Section 4 has several parameters. We use this case study to evaluate the influence of the parameters maximum path length, the solver to use, unchecked steps, maximum amount of test cases, and boundary value analysis on the runtime.

Manual Adaptation A few manual pre–processing is necessary in order to use the case study model with AcT: import the model, add *guard*s and *localPostcondition*s in OCL syntax, flatten the *LoopNodes*, and replace the *DataStoreNodes* by *Properties*.

The model was provided as XMI export from Atego® Artisan Studio. Due to slight differences in the implementation it is not directly possible to load an XMI file from Atego® in Eclipse. We manually removed some objects not recognised by Eclipse and corrected some typing errors in the XMI file. Every *Action* and *ControlFlow* contains C code snippets, but no OCL constraints. We added *guards* and *localPostconditions* reproducing the semantics of the C code snippets contained in the original model. The function specifying which *ControlFlow* to take after each *Action* is well–defined and defined over the complete domain. The original model used local C struct variables modelled by the *DataStoreNodes* contained by the *Activity*. Our implementation can handle variables modelled as *Property*. 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 upon a change of the index. This may produce wrong behaviour but seemed good enough to evaluate the runtime of our algorithm. The *LoopNodes* contain further model elements in their *bodyPart*. We connect those elements from the *bodyPart* to additional *ControlFlow*s and *ActivityNode*s emulating the counter loop semantics. The *LoopNode* is discarded and its augmented *bodyPart* is directly embedded in the *Activity*.

Runtime Measurement Results. We examine the influence of the used solver, the maximum path length, the maximum number of test cases, the unchecked steps, and boundary value analysis on the runtime of our algorithm.

Different Solvers We found that the runtime grows almost exponentially with the maximum path length: It doubles when the maximum path length increases by 6-7. The solvers LPsolve, Cplex, and GeCoDE are equally fast. Couenne consumed about three times the runtime consumed by LPsolve. This is 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.

All constraints in the case study model are instances of mixed integer linear programming and the solvers implement effective methods for mixed integer linear programming. Consequently, we gain test data for every feasible path. The number of test cases grows exponentially with the maximum path length. For a maximum path length of 110 a total of 12,850,000 linear programs had to be solved to find 83,000 sets of test data. The fastest solver, LPSolve, took only 13 hours for this task. This makes 275 solved problems per second and among them 1.8 produced actual test data.

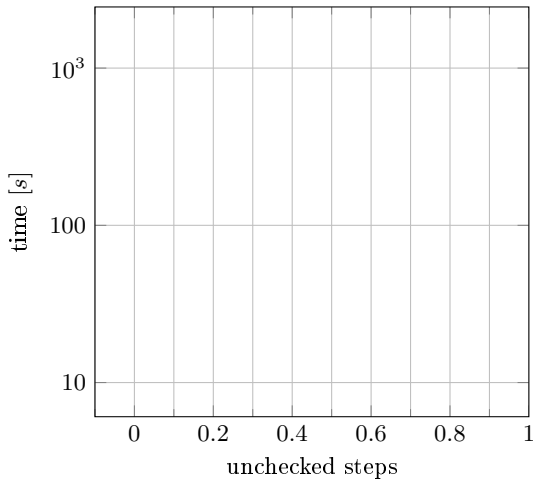

Fig. 4. Runtime of our implementation depending on the unchecked steps

Unchecked Steps In Figure 4, we plotted the overall runtime of our algorithm depending on the unchecked steps (see Section 4.4. We used Cplex as solver 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 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 — 2 — takes less than one hour, while it takes several days with unchecked steps set to 6.

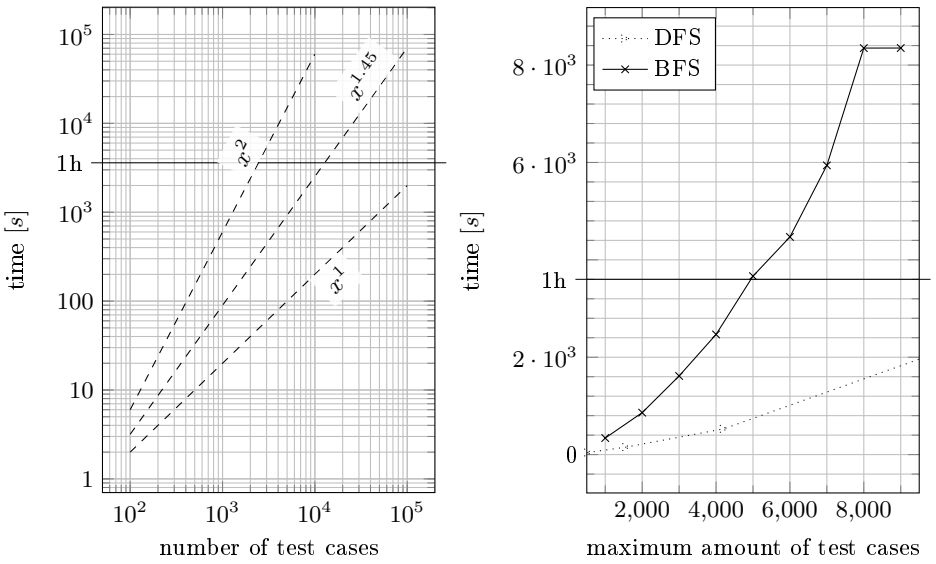

Fig. 5. Runtime depending on the number of test cases. Left: depth–first search. Right: breadth–first search

Maximum Amount of Test Cases The runtime as well as the number of abstract test cases grows exponentially with the maximum path length. The quotient of runtime and generated test cases grows polynomially. To illustrate that, we plotted the runtime of our algorithm against the number of test cases in Figure 5. For the plot on the left we did not use the parameter maximum amount of test cases but plotted the number of test cases actually found against the runtime of our algorithm using different values for maximum path length. It is logarithmic on both axes and we can see that the runtime grows with an exponent of 1.45.

Depth–first search without limit on the maximum path length but with a limit on the number of test cases, would not produce useful test cases. All generated abstract test cases would over and over share very long common sub–paths and some *ControlFlow*s would not be checked at all. Therefore, we implemented a breadth–first search–based alternative to the algorithm explained in Section 4.4. We use this alternative to evaluate the effect of the maximum amount of test cases on the runtime; for all other experiments we use only depth first search.

In Figure 5 right, we plot the runtime of our algorithm with breadth–first search. The dotted line shows a small section of the left plot. 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. In depth–first search, usually only a small amount of the constraints change between two subsequent solver invocations. For breadth–first search it happens more often that all constraints have changed between two subsequent solver invocations. Furthermore,

we see that the generation of 9,000 test cases took only slightly longer than the generation of 8,000 test cases. We can not explain this last effect.

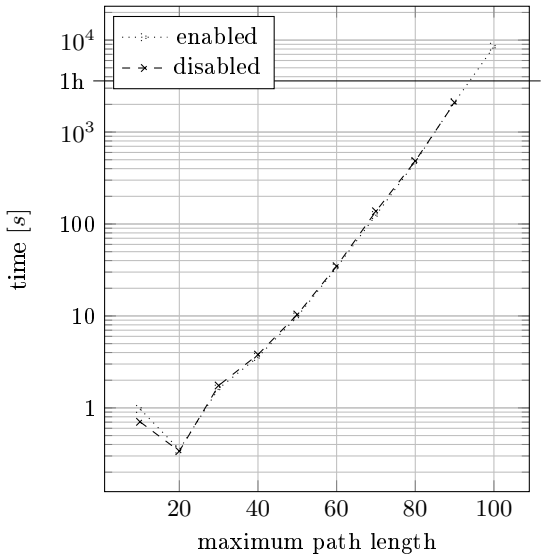

Fig. 6. Comparison of runtime with boundary value analysis enabled and disabled

Boundary Value Analysis Finally, we analysed the impact of boundary value analysis on the runtime of our implementation. As explained in Section 4.3, an additional objective has been added in the AMPL program. We minimise the sum of all variables in the initial state. In Figure 6, we show the runtime of our algorithm depending on the maximum path length. We use LPsolve as solver. The runtime with boundary value analysis and the runtime without boundary value analysis are plotted. The plot clearly shows that there is no difference in runtime between generating boundary test data or arbitrary test data.

This result is plausible because the number of solver invocations where extra work needs to be 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 that recognise a control flow path as infeasible. Consequently, only 4.5% of all solver invocations are actually performing some extra work. For larger maximum path lengths this fraction is declining further.

5.3 Limitations and Outlook

A natural limitation for constraint-solver-based test data generation is decidability. There is no effective method to generate variable instantiations that satisfy an arbitrary mathematical term and heuristics may not come to a halt on some problem instances. Currently, we support only a subset of the UML activity modelling elements. Hierarchical activity diagrams, complex data types, data flow, concurrency, and exception handling are currently not supported.

Namely the hierarchical modelling elements *StructuredActivityNode*, *CallBehaviourAction*, *CallOperationAction* are missing. These elements could be flattened by a preprocessing algorithm just as we did manually for our case study in Section 5.2. We support the data types integer, boolean and float. AMPL has support for arrays, so it is possible to add support for arrays of those basic types to our tool by adjusting the UML to AMPL transformation slightly. Further a variable with a complex data types such as *Class* holding integer, boolean, or float *attribute*s can be replaced by several basic typed *Property* elements. Currently we use only *Property* elements to represent a data store; of course *ObjectNode* and its subtypes could be handled in a similar manner or just be replaced by a set of *Property* elements during preprocessing; this is done manually in Section 5.2. The concurrency modelling elements *ForkNode* an *JoinNode* are not interpreted correctly, as our tool is not focused on concurrency. There is no provision to handle *ExceptionHandler* elements.

Finally, our path search algorithm currently suffers from combinatorial explosion. Generating test cases that adhere to a feasible coverage criterion like, for example, control flow coverage, would avert this problem. In [15] a comprehensive framework for generating test suites that adhere to model coverage criteria has been presented. At its core it uses a depth first search based algorithm that is supported by abstract interpretation. This search algorithm can be replaced by Algorithm 1, which is supported by symbolic execution and constraint solvers; this has been done by Amin Rezaee in [13]. Further some adaptations of the framework need to be done to work with *Activity* instead of *StateMachine*.

6 Summary and Recommendation

In this paper, we presented an efficient way to integrate state–of–the–art constraint solvers into a model–based testing tool. Depending on the specific needs of the modeller, a variety of constraints can be used in the model. We showed that the automatic test data generation is especially fast when constraints are specified in terms of a decidable theory and solved with an optimised solver. On the other hand we generated test data for models whose constraints are formulated as instances of an undecidable problem like, e.g., mixed integer non–linear programming. Although our approach makes it possible to generate test data for models with undecidable constraints, this can be very time consuming. Furthermore, it is not guaranteed that existing test data will be found. We recommend restricting oneself to linear inequalities or another decidable constraint formulation.

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. Using early infeasible path elimination with two unchecked steps massively reduces the runtime of our algorithm.

Finally, we showed that the presented approach allows 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.

References

1. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.: A Search-based OCL Constraint Solver for Model-based Test Data Generation. In: Quality Software (QSIC), 2011 11th International Conference on. pp. 41–50 (July 2011), http://dx.doi.org/10.1109/QSIC.2011.17
2. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and Bounds Tightening Techniques for Non–Convex MINLP. Optimization Methods and Software 24(4-5), 597–634 (2009), http://dx.doi.org/10.1080/10556780903087124
3. Berkelaar, M., Eikland, K., Notebaert, P.: lpsolve : Open source (Mixed-Integer) Linear Programming system, http://lpsolve.sourceforge.net/5.5/
4. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press, 2 edn. (2002), http://ampl.com/resources/the-ampl-book/
5. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on. pp. 139–150 (Nov 2004), http://dx.doi.org/10.1109/ISSRE.2004.12
6. Krieger, M.P., Knapp, A.: Executing Underspecified OCL Operation Contracts with a SAT Solver. Electronic Communications of the EASST 15 (2008), http://journal.ub.tu-berlin.de/eceasst/article/view/176
7. Kurth, F.: Automated Generation of Unit Tests from UML Activity Diagrams using the AMPL Interface for Constraint Solvers. Master's thesis, Hamburg University of Technology, Germany, Hamburg (January 2014), http://www.sts.tuhh.de/pw-and-m-theses/2014/kurth14.pdf
8. Laborie, P.: IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems. In: Hoeve, W.J., Hooker, J. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science, vol. 5547, pp. 148–162. Springer, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-01929-6_12
9. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., Guoliang, Z.: Generating Test Cases from UML Activity Diagram based on Gray–Box Method. In: Asia–Pacific Software Engineering Conference. pp. 284–291. IEEE Computer Society, Los Alamitos, CA, USA (2004), http://doi.ieeecomputersociety.org/10.1109/APSEC.2004.55
10. Malburg, J., Fraser, G.: Combining Search–based and Constraint–based Testing. In: International Conference on Automated Software Engineering. pp. 436–439. IEEE Computer Society (2011), http://dx.doi.org/10.1109/ASE.2011.6100092

11. Mingsong, C., Xiaokang, Q., Xuandong, L.: Automatic Test Case Generation for UML Activity Diagrams. In: International Workshop on Automation of Software Test. pp. 2–8. ACM (2006), http://dx.doi.org/10.1145/1138929.1138931
12. Object Management Group (OMG): OMG Unified Modeling Language™ (OMG UML), Superstructure (May 2010), http://www.omg.org/spec/UML/2.3/
13. Rezaee, A.: A New Approach to Optimized Generation of Test Cases using UML State Machine. Master's thesis, University of Isfahan, Iran, Isfahan (February 2014)
14. Tillmann, N., Halleux, J.: Pex-White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs, Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-79124-9_10
15. Weißleder, S.: Test Models and Coverage Criteria for Automatic Model–Based Test Generation with UML State Machines. Ph.D. thesis, Humboldt University Berlin (2010), http://model-based-testing.de/data/weissleder_phd_thesis.pdf
16. Weißleder, S., Sokenou, D.: Automatic Test Case Generation from UML Models and OCL Expressions. In: Software Engineering (Workshops)'08. pp. 423–426 (2008)