Scientia Iranica D (2017) 24(6), 3132–3147

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
www.scientiairanica.com

A novel approach to automatic model-based test case generation

A. Rezaee and B. Zamani*

MDSE Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran.

Received 24 May 2016; received in revised form 31 December 2016; accepted 17 July 2017

KEYWORDS
Model-based testing;
Automated testing;
UML state machine;
AMPL;
Constraint solver.

Abstract. This paper proposes a new method for automatic generation of test cases using model-based testing. As test model, class and state diagrams are used and constraints are expressed using Object Constraint Language (OCL). First, the state machine is converted into a mathematical representation in AMPL (A Mathematical Programming Language). Then, using a search algorithm and based upon coverage criteria, the abstract paths are selected from state machine. Second, using symbolic execution, the generated abstract path along with the constraints on this path is converted into the data of generated mathematical model. Third, the generated mathematical problem is solved with solvers that have interface with AMPL, and the test data are produced for each abstract test case. Finally, the generated test data and abstract paths are transformed into executable test cases. All-Transitions and All-States coverage criteria are used for conducting the search algorithm as well as the criteria to evaluate the quality of the generated test cases. To validate the work, by utilizing various solvers, the test cases are generated for various problems. The proposed technique is implemented as a tool, named MoBaTeG. The tool shows good result in terms of test case generation execution time, test goals satisfaction rate, source code instructions coverage, and boundary values generation.

1. Introduction

Given the complexity of today's computer systems, modeling is considered as a necessity, particularly in software engineering. Unlike traditional software development processes that focus on the code, in Model Driven Development (MDD), the main artifact is the model that drives the process. The final goal of MDD is to automatically build software from the model [1].

The basis of all model driven approaches is model transformation [2]. This means that the model with high level of abstraction is converted into another model with lower level of abstraction and, eventually, it is converted into code. Model transformation is the distinguishing factor between the traditional and the MDD approaches.

Software testing is one of the important parts of any software development process. Approximately, 50% of the project budget is spent on software testing [3]. Traditional testing is performed by a tester using manual approaches. Testing software systems in traditional manner is complex, time consuming, and error prone [4]. One of the systematic approaches to test case generation is Model-Based Testing (MBT). In MBT, the conformance of the System Under Test

*. *Corresponding author. Tel.: +98-31-37934537; Fax: +98-31-36699529*
E-mail addresses: a.rezaei@irisaco.com (A. Rezaee); zamani@eng.ui.ac.ir (B. Zamani)

doi: 10.24200/sci.2017.4528

(SUT) to the test model is evaluated. The test model can be created using one of the modeling languages, UML, automata, and petri net, to name a few [5].

In this work, a new method is presented for automatic generation of test cases using MBT techniques. In this approach, UML class diagram is used for modeling the static dimension of the system and UML state machine is used for modeling the dynamic behavior of the system. Constraints of the system are expressed using OCL. The OCL constraints can be used for defining the guards of state machine's transitions, state machine's state invariants, and the pre- and post-conditions of class diagram's operations. The operations can be used as the effect of UML state machine's transitions. The proposed method is comprised of several steps. First, the state machine is converted into a mathematical representation in AMPL (A Mathematical Programming Language (AMPL)). In this step, the model of AMPL is generated using some model transformations. Then, with a forward and depth-first search algorithm and based upon All-Transitions or All-States coverage criteria, the abstract paths from state machine are selected. Second, using symbolic execution, the generated abstract path and its element constraints are converted into the data of generated mathematical model. Third, the generated mathematical problem is solved with the state of the art and powerful solvers that have interface with AMPL and the test data are produced for each abstract test case. Finally, the generated test data and abstract paths are transformed into executable Java unit test.

It should be noted that each solver implements one (some) specific algorithm(s) or heuristic(s). These algorithms can be different in different solvers. Hence, each solver can solve only one (some) specific type(s) of problems. Mo-BaTeG can solve different types of problems using various solvers. Kurth [6] listed some of the AMPL solvers and the problem types they could solve. The following are examples of such solvers: Cplex, Minos, Gecode, Jacop, and Couenne.

In this work, All-Transitions and All-States coverage criteria are used to conduct the search algorithm as well as the criteria for evaluating the quality of the generated test cases. In order to increase error detection power of the algorithm, the boundary value analysis is used for boundary test data generation. In this research, by utilizing various solvers, the test cases are generated for various problems, e.g., linear, non-linear, decidable, and undecidable, that are defined by OCL constraints in the test model.

This paper is organized as follows. Section 2 covers the related works in the context of MBT and automatically generated test cases. Abstract test case generation using different algorithms as well as test data generation using different technologies will be discussed in this section. Section 3 describes the proposed approach for automatic generation of test cases as well as algorithm, data structures, and meta-model for building the tool called MoBaTeG (Model-Based Test Generator). MoBaTeG is developed based upon ParTeG (Both tools are available in http://parteg.sourceforge.net). In Section 4, we evaluate our proposed approach via several case studies. Also, MoBaTeG is compared with ParTeG. Section 5 contains the conclusion, in which we review the contributions of this research and address the future works and threats to validity.

2. Related works

This section addresses the related works in the domain of automatic generation of test cases. Utting [5] categorized different approaches of MBT based upon several criteria, as indicated in Figure 1. His taxonomy was based on the modeling language that was used for building the test model, different algorithms and techniques that were used for test case generation, and the ways that the test cases were executed against SUT. These criteria have been shown in Figure 1 as three main branches of taxonomy. Since our research is focused on test generation, we only consider the second branch in the figure.

As illustrated in Figure 1, the "Test Generation" branch has two sub-branches. The first one ("Test Selection Criteria") categorizes the coverage criteria, which are used as the basis for the selection of abstract test cases. The second one ("Technology") categorizes several technologies for concrete test data generation.

In the following sections, first, we describe the related works about abstract test case generations in Section 2.1. Then, the related works about concrete test data generation will be described in Section 2.2.

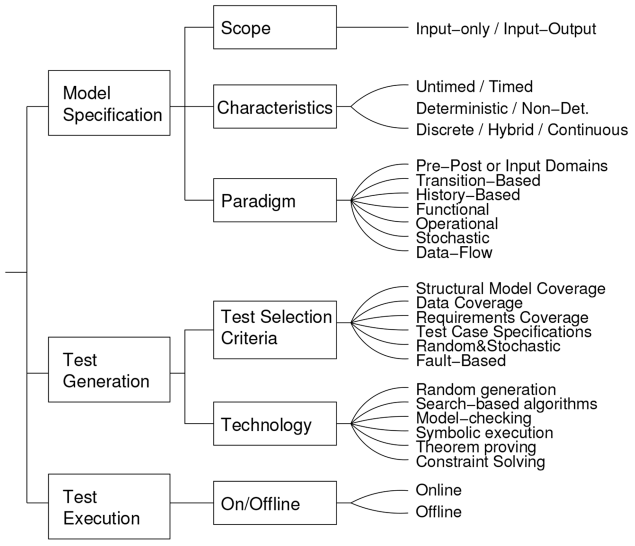

Figure 1. Utting's taxonomy of different MBT approaches [5].

Finally, the differences of our tool, MoBaTeG, with two other tools are mentioned in Section 2.3.

2.1. Abstract test case generation

The coverage criteria are used for conducting search algorithm as well as for evaluating the quality of generated test cases. After selecting a coverage criterion for test case generation, the abstract test case generation algorithm will be executed, which will try to satisfy the determined coverage criterion. Also, the coverage criteria satisfaction rate determines the quality of generated test cases by our approach.

Weißleder [7] proposed several coverage criteria that were based on data flow and model structure. In his proposed approach, each coverage criterion would be converted into some test goals. Then, the search algorithm would be used to satisfy the generated test goals. The search algorithm traversed the state machine's transitions with backward approach and considered guard conditions to satisfy them. Since brute force search of searching space was impossible, the data flow-based approach was employed for conducting backward search algorithm towards data definitions. The proposed approach was implemented as a free and open source tool called ParTeG.

We propose a forward search algorithm based upon state machine for abstract test case generation and test goals satisfaction. In our forward search algorithm, we traverse state machine in a forward manner and, in each state, check the satisfaction of unsatisfied test goals. In other words, our algorithm is run once and for each found path, it tries to satisfy the test goals. The backward search that is proposed in [7] is run once for each test goal. Furthermore, our proposed algorithm can support composite states and different ways through which the system can enter the composite state or exit it. However, some of the re-searchers proposed some approaches that did not consider composite state and only considered restricted parts of state machine, or did one additional step for flattening state machine before the execution of test case generation algorithm. We do not flatten state machines.

In [8,9], some of the searching graph algorithms are introduced as a technique for finding proper test cases. Kurth [6] used depth- and breadth-first search for abstract test case generation from UML activity diagram. Also, he introduced two parameterized termination conditions to ensure the termination of search algorithms. The user could identify the maximum amount of test cases to be generated as well as the maximum depth of model to be searched by search algorithms. He did not use coverage criteria and test goals for conducting search algorithm. We design and implement a depth-first search algorithm with forward approach on UML state machine. Furthermore, we use structural-based coverage criteria to conduct the search algorithm. The proposed search algorithm tries to satisfy test goals that are related to the selected coverage criteria.

2.2. Concrete test data generation

As indicated in the "Technology" branch in Figure 1, several technologies can be used for test data generation: random test data generation from feasible set, search-based test data generation, model checking, symbolic execution, theorem proving, and constraint solving. Random selection of test data cannot be considered as a proper method for test data generation, because it may not satisfy all the constraints within abstract path and the generated test data may be of low quality. However, it can be used for comparison with other approaches.

Weißleder and Sokenou [10] proposed a data-oriented approach for generating test data. In this approach, they used abstract interpretation and categorized input data in several parts based on the data domain. Then, the boundary value of each part would be selected as test data. In contrast to data partitioning for boundary test data generation, we use mathematical optimization for boundary value analysis. Also, we use symbolic execution for test data generation in contrast to data-oriented test data generation.

Ali et al. [11] defined some fitness functions based on OCL and generated test data using search algorithms. They used genetic algorithm and implemented their approach as an OCL-based constraint solver. However, up to this date, this solver has not been made publicly available. Malburg and Fraser [12] used a mixed approach. They used genetic algorithm for improving test data; also, for conducting genetic algorithm, they used special mutation operators performing dynamic symbolic execution. Their approach focused on white-box testing using the program's source code. Also, PEX [13] from Microsoft® could be used for white-box unit test. It used dynamic symbolic execution based on source code. Furthermore, for solving path conditions, it used a constraint solver. We use static symbolic execution and do not need source code for test case generation. Hence, our tool can also be used for black-box testing.

Krieger and Knapp [14] proposed a technique for converting OCL constraints into Boolean formulae and specified some states of system that could satisfy the constraints using SAT solvers. We use AMPL programming language and model OCL constraints in AMPL. Hence, our approach is not limited to one special heuristic or solver, and we are able to use several heuristics and solvers that have interface with AMPL. Each solver is able to solve one or some special types

of problems. We are able to solve different types of problems using different solvers.

In [6], Kurth proposed an algorithm for converting activity diagram, control flow path, and OCL constraints into mathematical model in AMPL modeling language. After generating AMPL model, the test data would be generated using solvers. His proposed approach was implemented as an open source tool called AcT (Activity Tester).

We use AMPL and solver-based technique, too. The difference between our work and [6] is that we use state machine as input model; however, in [6], activity diagram was used as test model. In [6], the OCL constraints could be used for defining the guard of control flows and post-conditions of each activity. In our work, OCL constraints can be used for defining the guard of transitions, pre- or post-conditions of class diagram's operations (these operations can be used as the effect of state machine's transitions), and states invariants. The semantics of these constraints are different from those used in [6]. Hence, the algorithm for converting them into AMPL is also different.

2.3. Summary

To summarize the properties of our tool, MoBaTeG, versus ParTeG [7] and AcT [6], we have prepared Table 1. In this table, three tools are compared from five aspects: test model, coverage criteria, abstract test case generation algorithm, test data generation, and boundary value analysis.

3. MoBaTeG: Model-Based Test Generator

In Model-Based Testing (MBT), after building the model, it will be used for test generation. Usually, the modeling is done in a manual process by modeler or tester using a modeling tool. Afterwards, the test cases will be generated automatically from test model using a model-based test generation tool. In this paper, the test generation problem is converted into a mathematical representation in AMPL language. Then, with a forward and depth-first search algorithm and based upon All-Transitions or All-States coverage criteria, the abstract paths from state machine are selected. Second, using symbolic execution, the generated abstract path and its element constraints are converted into the data of generated mathematical model. Third, the generated mathematical problem is solved with the state of the art and powerful solvers that have interface with AMPL and the test data are produced for each abstract test case. Finally, the generated test data and abstract paths are transformed into executable test cases. The overview of our approach is shown in Figure 2. There are four steps in this figure that will be described in Sections 3.1 to 3.4, respectively.

3.1. Normalization

This step accepts UML model as input and generates test case graph as out-put. The goal of this part is checking the correctness of OCL constraints. If all constraints are syntactically correct and supported by our approach, the abstract syntax tree will be generated for them. Also, we copy the required parts of UML input model and related OCL constraints in an internal structure, called test case graph. Then, we use this structure for test case generation and forget input UML model. Also, in this part, we transform the coverage criteria into test goals. All of these tasks are performed using ParTeG tool. It should be noted that we have modified internal structure of this tool in some parts so that it is able to support our approach's requirements. Partial meta-model of the test case graph is indicated in Figure 3.

The properties of TCGTransition class are those

Table 1. Comparing MoBaTeG with ParTeG [7] and AcT [6].

	ParTeG	**AcT**	**MoBaTeG**
Test model	UML state machine and class diagram	UML activity and class diagram	UML state machine and class diagram
Coverage criteria	Structural-based and data flow-based	No use	Structural-based
Abstract test case generation algorithm	Backward search	Backward and forward search	Forward search
Test data generation	Abstract interpretation	Symbolic execution	Symbolic execution
Boundary value analysis	Data partitioning	Mathematical optimization	Mathematical optimization

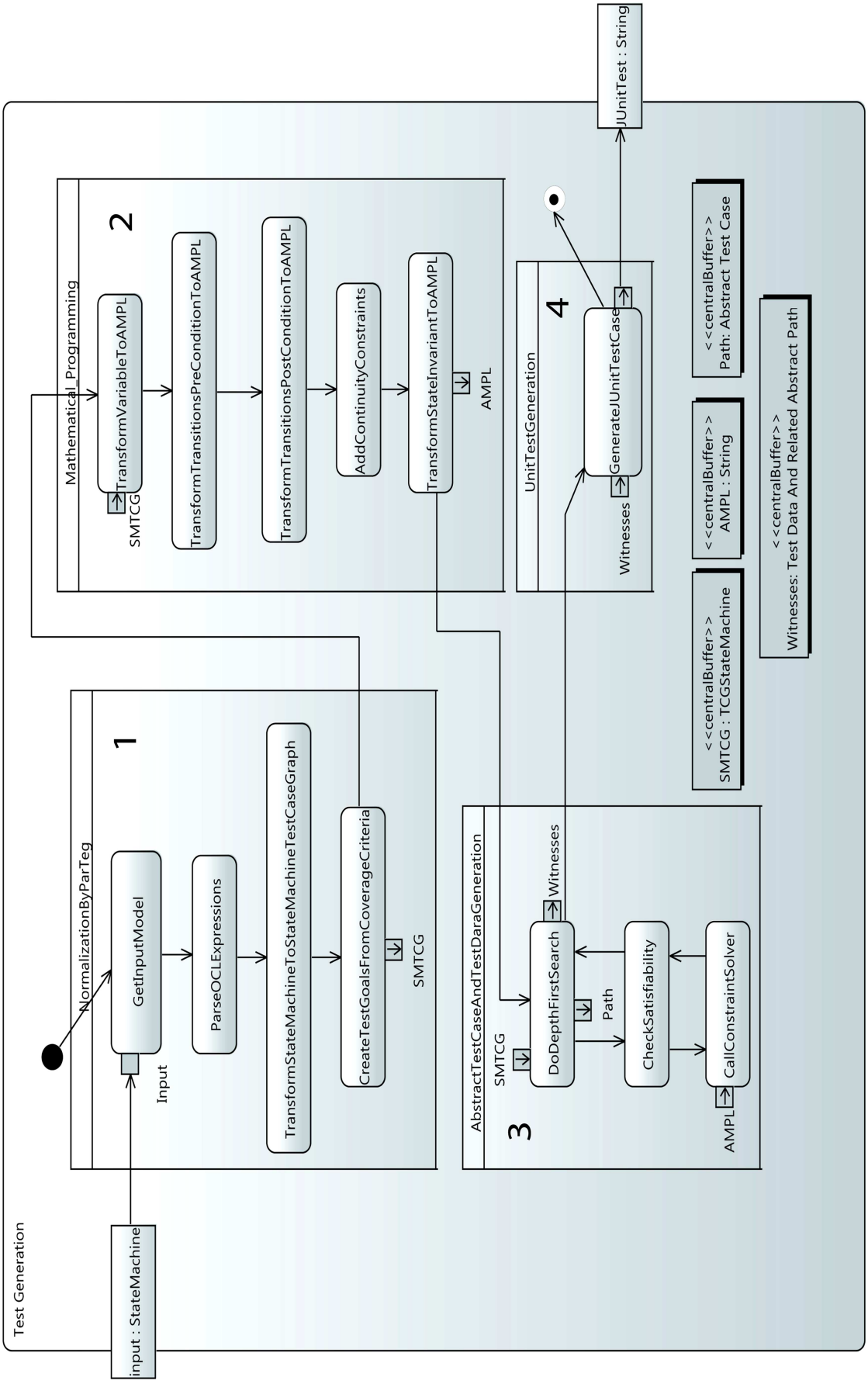

Figure 2. Test case generation process in the proposed technique.

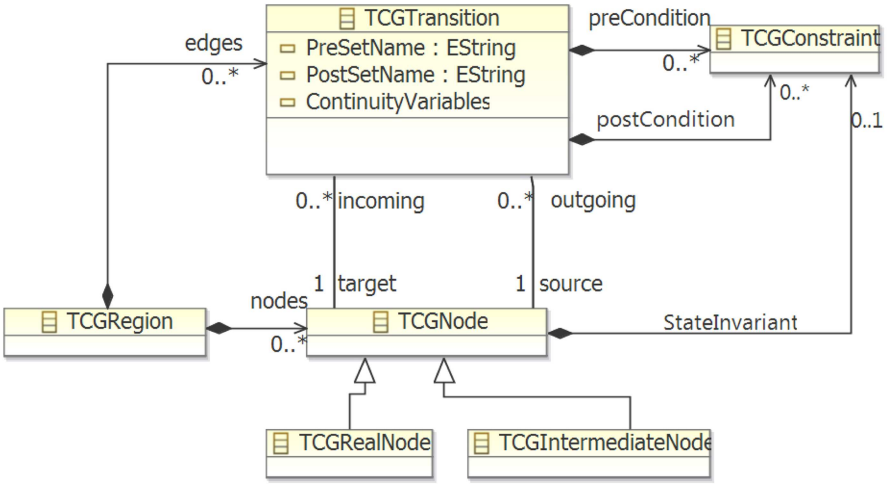

Figure 3. Partial test case graph meta-model (adapted from [7]).

properties that are added to the test case graph meta-model in this paper. Note that ParTeG does not support several pre- or post-conditions. However, in this approach, we support several pre- or post-conditions for each transition. The guard constraint of UML state machine transition and pre-condition(s) of the operation that are called in the effect of that transition, if exists, are converted into transition pre-condition(s) in the test case graph. Also, the post-condition(s) of the operation that is called in the effect of UML state machine transition, if exists, is (are) converted into transition post-condition(s) in the test case graph.

In the proposed approach, Integer, Real, and Boolean data types are sup-ported. Also, we support if-then-else statements for defining OCL constraints. Table 2 shows the supported operations in this approach.

3.2. Mathematical programming

In this step, we transform input model to mathematical representation in AMPL. This part accepts test case graph as input and generates AMPL model. In this section we describe the transformation of test case generation problem into mathematical and constraint satisfaction problem. As illustrated in Figure 2, this part consists of model variable to AMPL variable transformation, the pre-conditions of state machine's transition to AMPL constraints transformation, the post-conditions of state machine's transition to AMPL constraints transformation, continuity constraints addition to AMPL model, and state invariant to AMPL constraint transformation. Each of these steps is described in the following sections.

Table 2. Supported operations for defining OCL constraints.

Mathematical operations	−, +, *, /
Relational operations	<, <=, >, >=, =, <>
Boolean operations	and, or, not, Xor, Implies

3.2.1. Transforming test model variables to AMPL variables

In this step, all of the test model's variables involved in defining the OCL constraints are identified. For this purpose, we design and implement one model visitor that extracts all variables and parameters that are used in OCL constraints, and then, these variables will be transformed into AMPL variables.

The data structure that is used for holding variables is shown in Figure 4. The model visitor gets test case graph, traverses all transitions, and visits all states of it. If model visitor finds any constraints in any of these elements, it extracts the constraint and gives it to constraint visitor. The constraint visitor accesses atomic elements of constraints by traversing the abstract syntax tree of constraints with regard to constraint type. Then, with checking of these atomic elements, it recognizes the parameters and variables

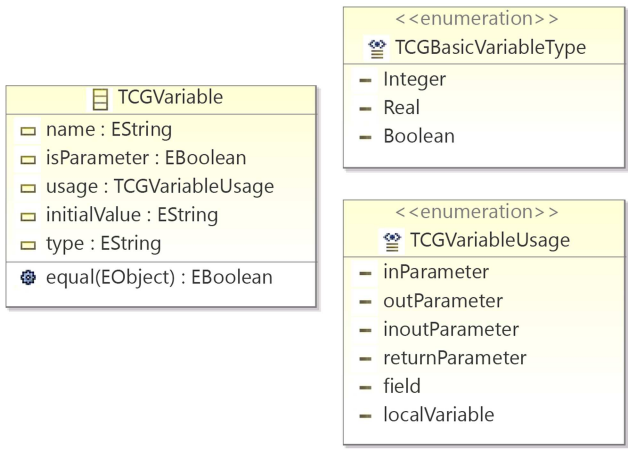

Figure 4. The data structure for storing data (from [6]).

and copies them into the data structure shown in Figure 4.

After determining the system's parameters and variables, it is necessary to convert them into AMPL. In the AMPL model, the type of each variable can be Integer, Real, or Boolean. The type of AMPL variables is Real by default. We need to define upper and lower bounds for Integer and Real variables. We set lower and upper bounds of these variables to -10000 and $+10000$, respectively. Without defining any bound for data types, some solvers cannot find correct answer to the problem in hand. Furthermore, we consider 0 and 1 for lower and upper bounds of Boolean data type that are equivalent to "false" and "true," respectively.

Since a variable can have different values in each level of execution, we define them as an array of variables with constant length, i.e. "pathlength," in AMPL. The "pathlength" is a parameter that is used in this paper for specifying the number of execution levels in one abstract path. This parameter is defined in model section of AMPL and its value is determined in data section of AMPL. If the "isParameter" property of TCGVariable is "true," then TCGVariable is a parameter. In this case, the value of TCGVariable is constant during execution, so we define it as a single variable.

3.2.2. Transforming pre-conditions of state machine transitions into AMPL

For each transition in state machine that has at least one pre-condition, we define one activation set with a random name and copy this name in "PreSetName" property of current transition. The activation set specifies in which level of execution the pre-condition must be considered. The activation set is defined in AMPL model as a subset of 0, ..., Pathlength. The activation set is empty by default.

The pre-conditions of current transition are defined as an indexed set of constraints on activation set. In AMPL, it is necessary for constraint to have a unique name. We use a random name with "_pre" sufix and one number that is specified with one counter as the name of each pre-condition. Also, we use constraint visitor for traversing abstract syntax tree of each pre-condition and extract pre-condition as string from related abstract syntax tree. For each atomic constraint in abstract syntax tree, if the atomic constraint is a parameter constraint, we only need to copy the name of this parameter in AMPL model. If the atomic constraint is a variable, we need to specify the index of variable. For this purpose, if the variable has not "@pre" label, we need to access the value of the variable in current level of execution, and if the variable has "@pre" label, we need to access the value of variable in the previous level of execution.

3.2.3. Transforming post-conditions of state machine transitions into AMPL

The transformation of transitions' post-conditions is similar to the transformation of pre-conditions. The only difference is that for some transitions, we need to add continuity constraint. Adding continuity constraints is described in the next section.

3.2.4. Adding continuity constraints to AMPL model

In transition's pre-conditions, the values of variables are determined based upon the current execution level. If the transition has some post-conditions, the execution level of algorithm increases by one unit and the constraint must be evaluated based upon the values of variables in the new level of execution, except when the variable has "@pre" label. For each variable that participates in the post-conditions and does not have "@pre" label, a new value must be set so that all constraints in the current execution level are evaluated as "true." If some of the model variables do not participate in describing the post-conditions, or all the participated variables have "@pre" label, we need to add continuity constraint for them. The continuity constraint says that the variable must hold its previous value. For example, for variable "x", the continuity constraint is "x=x@pre" in OCL language. The AMPL equivalent to this constraint is "$x[i] = x[i-1]$". Each continuity constraint is defined as an indexed constraint on the activation set of transition's post-conditions ("PostSetName"). To ensure the uniqueness of constraint's name, we use "PostSetName" property of current transition with "_post" sufix and a number that is specified with a counter.

3.2.5. Transforming states invariant into AMPL constraints

In UML state machine, each state can have a state invariant. While the system is in one state, the state invariant must be "true." For converting state invariant into AMPL constraint, the implemented algorithm gets test case graph as input and meets all states. If the state is instance of "TCGRealNode," it can have state invariant. If the "TCGRealNode" has a state invariant, this constraint is transformed into AMPL constraint. If the state does not have name, one random name is assigned to it. This name will be used for the name of state invariant's activation set. Then, the constraint is defined as an indexed set on activation set. The name of the constraint is built by adding the "_invariant" suffix to the name of the state.

3.3. Abstract test cases and concrete test data generation

Abstract test cases are some paths in the test model that are selected according to some specific criteria. We use structural coverage criteria for conducting test generation algorithm. After the test goals are

generated according to these coverage criteria using ParTeG, we use one depth-first and forward search algorithm to extract abstract path from test model. Each abstract path satisfies at least one test goal. The search algorithm that is designed and implemented in this paper starts the execution from the initial state of the state machine and traverses state machine's transitions in depth using forward search approach. By visiting each state, the algorithm checks whether any of the test goals can be satisfied according to the abstract path or not. Furthermore, the satisfiability of constraint within path is checked. For this purpose, the current abstract path is converted into AMPL data using symbolic execution. Then, we try to solve the AMPL model using different state of the art and powerful solvers that have interface with AMPL. If the solver cannot solve the constraint within the abstract path, that path is recognized as an "Infeasible Path," and the search algorithm must do backtrack and continue the execution in another unvisited path. If the solver can solve all the constraints within the path, the search algorithm can continue execution in that path in depth. Furthermore, if this abstract path can satisfy at least one test goal that is not satisfied yet, the data generated by the solver will be used as test data.

To ensure that the algorithm terminates and is not trapped in an infinite loop, the algorithm uses some fuel as search limit. The fuel determines how many transitions the algorithm is allowed to traverse. If the fuel of algorithm is finished in some state, the search algorithm does backtrack and continues the execution in another unvisited path. For each backtrack, the fuel of algorithm is increased by one unit.

The search algorithm supports composite states and different ways that the system can enter composite state or exit it. We consider three different ways for entering the composite state as well as exiting it. In the following, the three ways for entering the composite state have been described.

1. The target state of transition is an entry point and with another transition, the system enters a composite state;
2. The target state of transition is a composite state and after that, the system must continue the execution for each initial state of composite state's regions;
3. The source of transition is outside and the target of transition is inside the composite state.

Also, three ways for exiting composite state are as follows:

1. The target state of transition is exit point and by another transition, the system exits the composite state;
2. The target state of transition is final state of the composite state and after that, the system must continue the execution for each transition for which the composite state is source state.
3. The source of transition is inside and the target of transition is outside the composite state.

We consider and moderate all of these ways for entering or exiting the composite state. We describe three features of our algorithm for abstract path and test data generation in the next three sections.

3.3.1. Infeasible path detection

One of the important features of the presented algorithm is the capability of infeasible path detection. The infeasible path means that there is not any data for satisfying all of the constraints within the abstract path. For example, if the abstract path contains $a \geq 12$ and $a \leq 5$ as constraints and there is not any post-condition between these two constraints that change the value of a, these constraints cannot be satisfied; thus, this abstract path is recognized as "Infeasible Path" and the search algorithm must do backtrack. With detection of such paths and preventing search algorithm to progress in these paths, most of the search space can be eliminated, which leads to lower execution time of the algorithm.

As an example, consider the state machine in Figure 5, which includes an infeasible path indicated by the red color. This path is infeasible since there is not any test data to satisfy it. Each path needs to start with any value for variable "a" that is greater

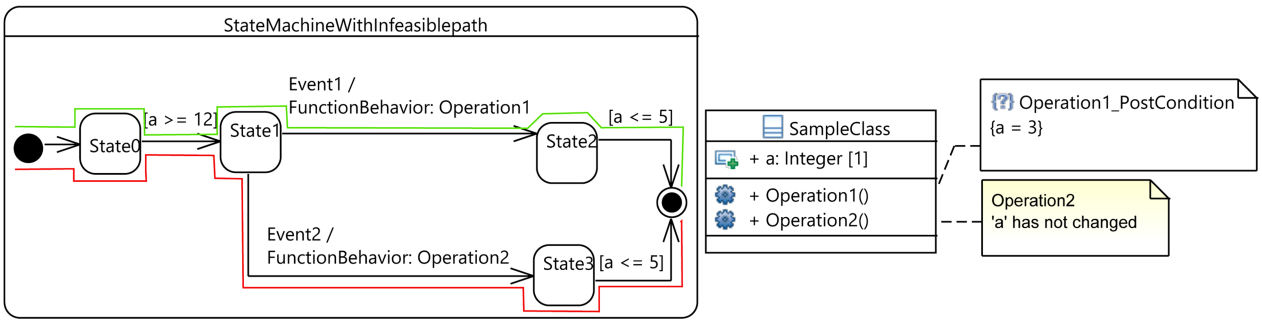

Figure 5. Feasible and infeasible paths.

than or equal to 12. With this value, the transition from "State0" to "State1" can be satisfied. If "Event1" occurs, the third transition of green path will be satisfied and "Operation1" will be run as an effect of transition. Calling "Operation1" will set variable "a" to 3 based upon post condition of "Operation1". However, the occurrence of "Event2" leads to traversing the third transition of the red path and results in calling "Operation2". In this case, the value of variable "a" has not changed. The guard condition of the fourth transition in both paths (red and green) is defined as "$a <= 5$". In the third transition of the green path, the value of variable "a" is set to 3. Hence, the fourth transition can be satisfied. On the other side, the third transition of the red path does not lead to change in the value of variable "a" and it keeps its previous value ($a >= 12$). Therefore, the guard condition of the fourth transition cannot be satisfied. This is why the red path is an infeasible path.

If it is impossible to satisfy a path, all of the extensions of that path will be infeasible, too. With increase in the depth of search path, the number of detected abstract paths will grow exponentially. For one large state machine that each state has two outgoing transitions and the depth of state machine is 20, the total number of abstract paths will be 1048576. If one abstract path with length 6 is recognized as an infeasible path, 16384 abstract paths will be removed from the search space. Thus, with "Infeasible Path Detection" feature of search algorithm, we can get rid of searching for many useless paths.

3.3.2. Warm start

Warm start is one of the features of AMPL programming language and most of its solvers. If a problem has been solved several times and in each execution different data are used, the knowledge that is gained in previous executions can be utilized and the problem is solved faster; this feature is called "Warm Start." When using "Warm Start" feature, the AMPL model must be fixed and the AMPL data must change in each execution. We do this with transforming state machine to AMPL model and for each abstract path, we generate AMPL data using symbolic execution. Therefore, the AMPL model will be fixed and the AMPL data will be reset in each execution.

Also, if the solver has been changed between the executions or the solver cannot gain any knowledge from previous executions, AMPL saves the last solution. Many of the algorithms can use last solution as a starting point. Furthermore, in this paper, by utilizing depth-first search algorithm, the newly found abstract path is extension of the last found path in most cases. Hence, the previous solution can be very useful because the new path only has a few additional constraints compared to the last found path.

3.3.3. Boundary value analysis

Any data that satisfy all of the constraints within abstract path can be used as test data. However, data that are at the boundary of feasible set can be more useful than other data inside this set. Therefore, we use boundary value analysis for boundary data generation in our work.

If all constraints of test model are linear, using simplex algorithm leads to automatic generation of boundary values. Simplex algorithm is one of the famous algorithms, which is used by many solvers that solve linear problems. However, if some of the constraints of test model are non-linear and the problem does not include any objective function, as soon as the solver finds a solution, it will return the solution. This solution does not guarantee the generation of boundary test data. Hence, we use one proper objective function that leads to boundary data generation before calling the solver.

3.4. Unit test generation

As described in Section 3.3, the abstract paths are generated according to coverage criteria and the test data are generated for each abstract path using AMPL programming language and related solvers. The generated abstract test cases as well as concrete test data are stored in data structures. Figure 6 shows this data structure.

As shown in Figure 6, a test suite may contain zero or more test cases and each test case contains one abstract path as well as zero or more variables with their values. Furthermore, each abstract path contains several transitions. Each transition has one target state and may have several pre- and post-conditions. Also, each transition may or may not have some events with or without some parameters. Furthermore, each state may or may not have one state invariant. Test suite is associated with SUT, which has two properties that specify the class and package on which the test must be executed. Finally, the function that handles the events is specified in SUT class.

4. Evaluation of MoBaTeG

The proposed approach for test case generation is implemented as an Eclipse-based and open source tool, named MoBaTeG, using Java programming language. We use five case studies for evaluating the MoBaTeG tool. Using different solvers is one of the main advantages of our work. The evaluation framework is described in Section 4.1, and the evaluation of MoBaTeG is given in Section 4.2.

4.1. Evaluation framework

We have evaluated MoBaTeG from different aspects using five case studies. All of the evaluations have been executed on a computer with Intel® core™ 2

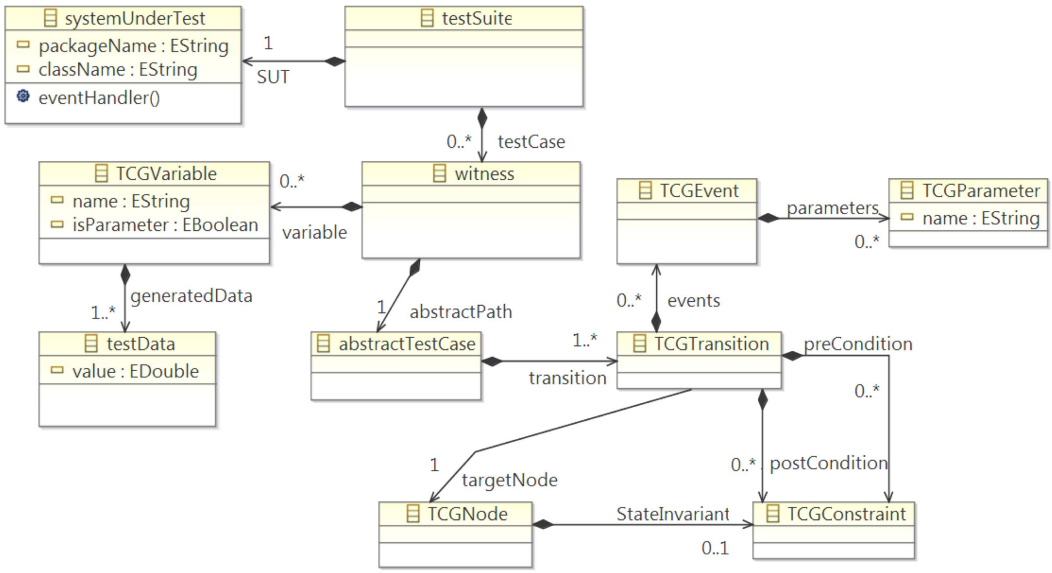

Figure 6. Unit test meta-model.

Duo processor, 4 GB memory, and the 64-bit version of Microsoft® Windows 7 operating system. In the following sections, we will describe the evaluation criteria.

4.1.1. Test case generation execution time

The test case generation execution time is one of the main criteria for automatic testing of software systems. Lower test case generation execution time can decrease test case generation cost. The execution time of the presented approach depends upon two main criteria, including the required time for solving a problem and the number of problems to be solved.

Test case generation execution time is calculated by different solvers. Test case generation algorithm will be executed 20 times for each coverage criterion and for each solver. Then, the average test case generation execution time will be calculated.

4.1.2. Test goals satisfaction rate

The coverage criteria can be used as the main criteria for evaluating the quality of generated test cases, i.e. the percentage of test goals satisfaction using different solvers will be used for evaluating the quality of generated test cases. We have used two kinds of coverage criteria, All-Transitions and All-States.

The quality of generated test cases that will be evaluated by this criterion depends upon the algorithm that is designed for abstract test case generation, the algorithm for transforming test case generation problem into mathematical problem, and the ability of the used solver for solving the generated mathematical problem.

4.1.3. The coverage of source code instructions

The coverage of source code instructions is another criterion that can be used for evaluating the quality of generated test cases. Higher coverage of source code instructions indicates that the quality of generated test cases is better. EclEmma [15] is used for calculating the coverage of generated test cases against source code. EclEmma is a tool for calculating the coverage of Java source code. In this evaluation criterion, source code instructions refer to Java Bytecode instructions.

4.1.4. Comparing with ParTeG

ParTeG is the only available MBT tool that uses UML class diagram and UML state machine annotated with OCL constraints as test model. Hence, we will compare our tool MoBaTeG with ParTeG. The comparison between MoBaTeG and ParTeG will be done based on "Test Case Generation Execution Time," "Test Goals Satisfaction Rate," and "The Coverage of Source Code Instructions" criteria.

4.1.5. Boundary value analysis

Generating boundary values will increase the quality of generated test cases and fault detection power of such test cases. Hence, the boundary values of generated test data will be evaluated.

4.1.6. Using different problems

A good solution must be able to solve different problems in terms of complexity. Therefore, ability of the proposed approach for test case generation will be evaluated using five different problems, which differ in terms of complexity. We will use linear and non-linear problems as well as decidable and undecidable problems.

4.2. Case studies

Five case studies have been used in this section for evaluating our tool, Mo-BaTeG. The first two problems are two versions of a triangle classifier. The classifier gets three numbers as the sizes of triangle's edges. Then, it

classifies the triangle under one of the titles: "invalid," "scalene," "isosceles," and "equilateral." For modeling the first problem (version one of the classifier), we do not use any Boolean operator in the definition of OCL constraints. For the second problem (version two of the classifier), we use a Boolean operator in the definition of OCL constraints.

The motivation for using triangle classifier with and without Boolean Operations is that the generated mathematical problems related to these two case studies are classified into two different kinds of problems in terms of complexity. Generated mathematical problem of "Triangle Classifier with Boolean Operations" is a sample of Satisfiability Modulo Theories (SMT) problems and the generated mathematical problem of "Triangle Classifier without Boolean Operations" is a sample of Integer Linear Programming (ILP) problems. Each of these problems needs different effort for solving. Hence, generating test cases using these case studies in our work shows the ability of our approach for solving these two different kinds of problems.

The third problem models the behavior of an air pump and shows the relationship between "air," "temperature," "volume," and "pressure." Ideal gas equation has been used for describing the relation between these variables. The class diagram and its operations as pre- and post-condition(s) of this case study are shown in Figure 7. The generated mathematical problem related to this case study is a sample of Mixed Integer Non-Linear Programming (MINLP) problems.

The fourth problem models the behavior of an ATM system. This case study contains 4 composite states. 2 out of 4 composite states have entry point pseudo state for entering the composite state. To generate test data for ATM case study, a sample of SMT problems needs to be solved.

The last problem models the behavior of Continuous Casting Machine (CCM). CCM runs the process whereby molten steel is solidified into a billet, bloom, or slab.

Table 3 shows the specifications of our case studies. The first two problems are samples of linear and decidable problems. The third problem is neither linear nor decidable. The fourth and fifth problems are both linear and decidable. In the state machines of case studies, the numbers of states are 17, 10, 20, 47, and 25, respectively. Also, the numbers of transitions are 30, 12, 21, 60, and 29, respectively. The test data generation problem complexity for all of the generated

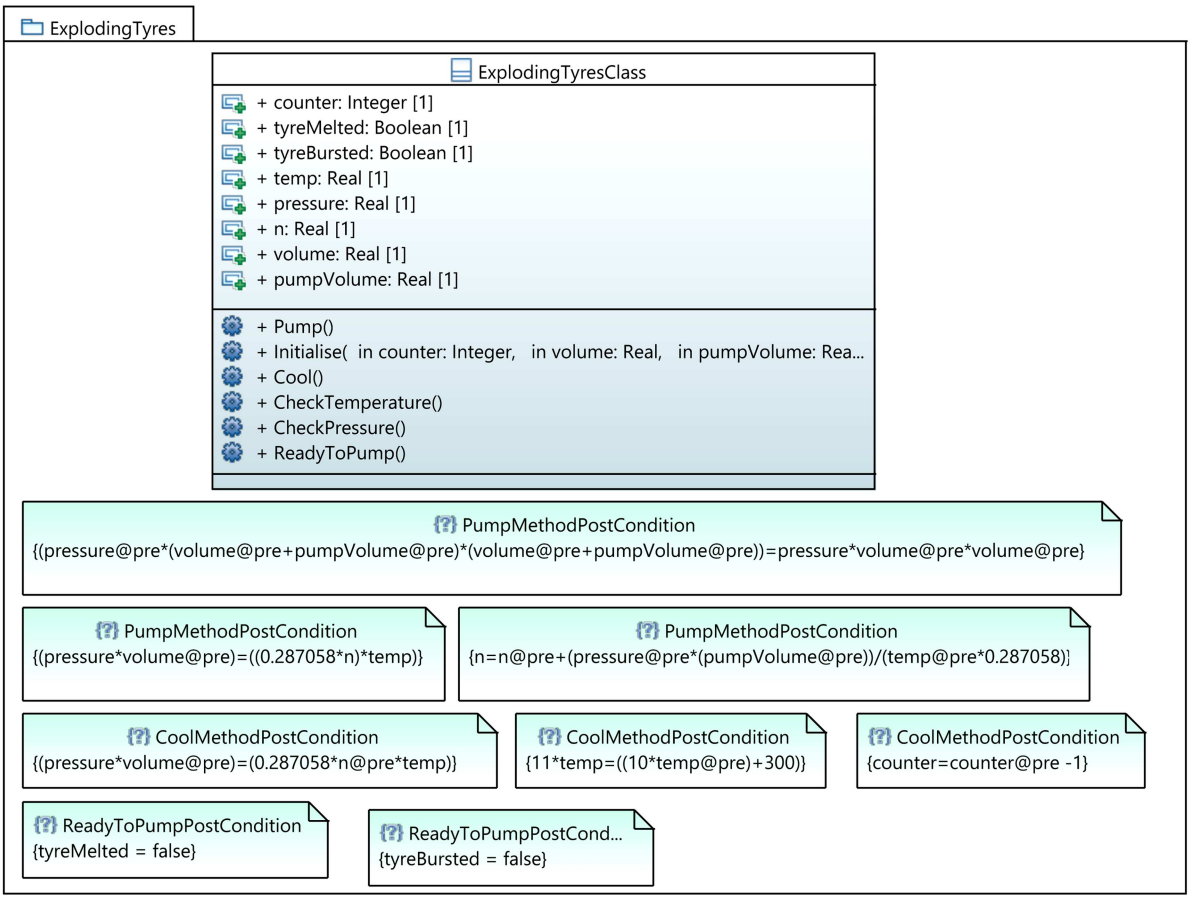

Figure 7. The class diagram and operations pre- and post-conditions.

Table 3. Specifications of five case studies used for evaluating MoBaTeG.

#	Problems	Linear	Decidable	Number of states	Number of transitions	Problem type	Problem complexity	Used data type
1	Triangle classifier without Boolean operations	Yes	Yes	17	30	ILP	NP-hard	Integer
2	Triangle classifier with Boolean operations	Yes	Yes	10	12	SMT	NP-hard	Integer
3	Air pump	No	No	20	21	MINLP	NP-hard	Integer, Real, Boolean
4	ATM	Yes	Yes	47	60	SMT	NP-hard	Integer, Boolean
5	CCM	Yes	Yes	25	29	ILP	NP-hard	Integer, Boolean

mathematical problems related to case studies is NP-Hard. In the first two problems, only Integer data type is used. Whereas, the third problem uses three data types in its model, namely, Integer, Real, and Boolean. Finally, two problems use Integer and Boolean data types for modeling.

4.2.1. Test case generation execution time

The average execution time of test case generation for both MoBaTeG and ParTeG tools has been calculated based on 20 executions. We consider All-Transitions and All-States coverage criteria. The average of execution times has been shown in Table 4. Two solvers have been used for calculating the average execution time using MoBaTeG tool for each problem.

For the first problem, the execution time of ParTeG is better than that of both of the solvers performed by MoBaTeG. In the next section, we will see that for the first problem, ParTeG cannot generate any test data. This is why the execution time of this tool is low. For the second problem, the execution time of ParTeG is significantly higher than that of MoBaTeG for both solvers. In the class diagram of the third problem, there are some operations that have several post-conditions. As mentioned in Section 3.1, ParTeG does not support several pre- or post-conditions. Thus, for this problem, ParTeG considers only one of the post-conditions of each operation. Despite this, ParTeG cannot generate any test case for the considered constraints. But, our tool, MoBaTeG, generates test cases within appropriate execution time. Therefore, this problem shows the superiority of MoBaTeG over ParTeG.

In the fourth case study, we use GeCoDE and

Table 4. The test case generation execution time of MoBaTeG and ParTeG.

#	Problem	Execution time (ms) of MoBaTeG			Execution time (ms) of ParTeG	
		Solver	All States	All Transitions	All States	All Transitions
1	Triangle classifier without Boolean operations	LpSolve	312	329	171	241
		Minos	307	363		
2	Triangle classifier with Boolean operations	GeCoDE	242	526	15484	15849
		JaCoP	2019	2366		
3	Air pump	Bonmin	8117	8291	Not supported	Not supported
		Couenne	9537	11897		
4	ATM	GeCoDE	1667	1739	Not supported	Not supported
		IlogCP	221	171		
5	CCM	GeCoDE	1660	1655	Not supported	Not supported
		JaCoP	28679	30207		

IlogCP for solving mathematical problem and generating test data. ParTeG cannot generate any test cases for this case study. When using All-State coverage criteria, the execution time of MoBaTeG using GeCoDE and IlogCP solvers is 1667 (ms) and 221 (ms), respectively. These values are 1739 (ms) and 171 (ms) when using All-Transitions coverage criteria, respectively. Therefore, MoBaTeG generates test cases in appropriate time.

In the last case study, when using All-State coverage criteria, the execution time of MoBaTeG using GeCoDE and JaCoP solvers is 1660 (ms) and 28679 (ms), respectively. These values are 1655 (ms) and 30207 (ms) when using All-Transitions coverage criteria, respectively. ParTeG cannot generate any test cases for this case study, too. The superiority of MoBaTeG over ParTeG is again shown in this evaluation.

4.2.2. Test goals satisfaction rate

Test goals satisfaction rate has been calculated for both MoBaTeG and ParTeG tools as well as for each problem and coverage criterion. This evaluation reflects the power of the abstract test case generation algorithm, the proposed algorithm for transforming test case generation problem into mathematical ones, and the solvers to solve the generated mathematical problems.

The test goals satisfaction rate for all of the case studies has been shown in Table 5. We have used two solvers for solving each problem. The first problem is solved by using MoBaTeG tool with both LpSolve and Minos solvers. This evaluation shows 100% satisfaction for both All-States and All-Transitions coverage criteria. However, these values for ParTeG tool are 12.5% and 7.14%. For the second problem, when MoBaTeG is used for test case generation and the selected solver is GeCoDE, all of the test goals that are generated using All-Transitions and All-States will be satisfied (i.e., the result is 100%). However, using All-States and All-Transitions coverage criteria for MoBaTeG with JaCoP solver leads to 87.5% and 90.90% satisfaction, respectively. These values are 100% and 90.90% for ParTeG. In this problem, MoBaTeG and ParTeG have similar behavior. While ParTeg cannot generate any test cases for the third problem, MoBaTeG satisfies all of the generated test goals for both coverage criteria when using Couenne solver. The results for All-States and All-Transitions coverage criteria for MoBaTeG tool with Bonmin solver are 85.71% and 100%, respectively. As we mentioned in Section 4.2.1, ParTeG cannot generate any test cases for the fourth case study (ATM). Therefore, the test goal satisfaction rate of ParTeG is 0% for both of All-States and All-Transitions coverage criteria. However, MoBaTeG satisfies 92% of test goals using All-State coverage criteria with both solvers. Test goal satisfaction rate of MoBaTeG when using GeCoDE and IlogCP for All-Transition coverage criteria is 79% and 20%, respectively. In the last case study, using MoBaTeG tool with both GeCoDE and JaCoP solvers results in 100% satisfaction for All-States coverage criteria and 93.1% satisfaction for All-Transitions coverage criteria. However, ParTeG cannot generate any test cases.

4.2.3. The coverage of source code instructions

After the test cases have been generated, they are run against source code and the source code instructions

Table 5. The test goals satisfaction rates of MoBaTeG and ParTeG.

#	Problem	Test goals satisfaction rate of MoBaTeG			Test goals satisfaction rate of ParTeG	
		Solver	All-States	All-Transitions	All-States	All-Transitions
1	Triangle classifier without Boolean operations	LpSolve	100%	100%	12.5%	7.14%
		Minos	100%	100%		
2	Triangle classifier with Boolean operations	GeCoDE	100%	100%	100%	90.90%
		JaCoP	87.5%	90.90%		
3	Air pump	Bonmin	85.71%	100%	Not supported	Not supported
		Couenne	100%	100%		
4	ATM	GeCoDE	92%	79%	Not supported	Not supported
		IlogCP	92%	20%		
5	CCM	GeCoDE	100%	93.1%	Not supported	Not supported
		JaCoP	100%	93.1%		

Table 6. The coverage of source code instructions using MoBaTeG and ParTeG.

#	Problem	MoBaTeG		ParTeG	
		All States	**All- Transitions**	**All- States**	**All- Transitions**
1	Triangle classifier without Boolean operations	102 out of 110 92.7%	106 out of 110 96.4%	15 out of 110 13.6%	83 out of 110 75.5%
2	Triangle classifier with Boolean operations	138 out of 138 100%	138 out of 138 100%	138 out of 138 100%	138 out of 138 100%
3	Air pump	209 out of 212 98.6%	212 out of 212 100%	Not supported	Not supported
4	ATM	229 out of 258 88.8%	229 out of 258 88.8%	Not supported	Not supported
5	CCM	489 out of 582 84%	558 out of 582 95.9%	Not supported	Not supported

coverage will be calculated using "EclEmma" tool. The results for "Coverage of Source Code Instructions" for all case studies are shown in Table 6.

The source code of the first problem consists of 110 instructions. When the test cases that are generated using MoBaTeG with All-States coverage criterion are run against source code, 102 out of 110 instructions are covered (92.7% coverage). The value of this criterion when using ParTeG with All-States coverage criterion is 15 out of 110 instructions (13.6%). When All-Transitions coverage criterion is used for generating test cases for the first problem, 106 out of 110 instructions are covered using MoBaTeG (96.4%) and 83 out of 110 instructions are covered using ParTeG (75.5% coverage). This problem shows the superiority of MoBaTeG over ParTeG.

The second problem's source code consists of 138 instructions. For both coverage criteria when using each of the tools, the generated test cases cover all instructions of the source code. Hence, in this problem, both tools have the same behavior.

The third problem's source code consists of 212 instructions. When the generated test cases by MoBaTeG and All-States coverage criterion are run against source code, 209 out of 212 instructions are covered (98.6% coverage). The result of using All-Transitions coverage criterion and MoBaTeG is 100%. As mentioned earlier, ParTeG cannot generate any test cases for the third problem.

The fourth problem's source code consists of 258 instructions. Running the generated test cases by MoBaTeG against source code covers 229 out of 258 (88.8%) source code instructions. This value is equal for both All-State and All-Transition coverage criteria. Running the generated test cases by MoBaTeG against the source code of the last problem results in 84% (489 out of 582) coverage for All-State coverage criterion and 95.9% (558 out of 582) coverage for All-Transition coverage criterion. As mentioned earlier, ParTeG cannot generate any test cases for the fourth and fifth problems.

4.2.4. Boundary value analysis

In this work, the boundary value analysis is performed for boundary data generation. As mentioned in Section 3.3.3, if all of the model's constraints are linear, using linear solvers can generate boundary test data. But, if the model's constraints are non-linear, using proper objective function can generate boundary test data.

As can be seen in Table 3, the first, second, fourth, and fifth problems are linear. Hence, using the proposed approach must generate boundary test data automatically. The results of our experiments in this section prove this claim. The third problem is non-linear. Thus, we use a proper linear objective function and send it to AMPL. Then, we check the generated test data for some abstract paths. We see that all generated test data are boundary data.

5. Conclusion

This paper proposed an algorithm for automatic generation of test cases from UML class diagrams and UML state machines annotated with OCL constraints as well as executable Java unit tests. The detailed elaboration of tasks and subtasks for test case generation from UML models is given in Section 3. The main part of these tasks is the transformation of test model and its constraints into the corresponding model in a mathematical programming language, AMPL, and using solvers for solving the model. Section 4 was dedicated to evaluation of the proposed algorithm via several case studies. In the following, we first review

the contributions of this research and then discuss the future works and threats to validity.

5.1. Contributions

In this paper, a new depth-first and forward search algorithm is proposed, which is based on UML state machine and aims at satisfying test goals that are generated form coverage criteria. The abstract test cases are generated using this search algorithm. The proposed forward search algorithm is run once and for each abstract path that is found, satisfaction of unsatisfied test goals is checked. Instead, ParTeG runs backward search one time for each test goal. For example, for 20 test goals, ParTeG needs to run its back-ward search 20 times, whilst MoBaTeG runs its forward search algorithm only once.

MoBaTeG supports composite states and all the possible ways for entering and exiting such states. However, some other approaches do not support composite states or they need to flatten state machines before the execution of test case generation algorithm, which is an extra task. Also, they may support only some of the elements of UML state machine. As an example, ParTeG does not support entry and exit point pseudo states, while MoBaTeG supports these pseudo states.

To prevent search algorithm from traversing infinite paths or getting trapped in loops that may exist in the state machine, we consider fuel for our search algorithm. This fuel is equal to the number of transitions that the search algorithm is allowed to traverse. If the fuel of algorithm is finished, it must do backtrack and continue its search in another path that it has not visited so far. With each backtrack, the fuel of algorithm will be increased by one unit. This feature of MoBaTeG prevents search algorithm from getting trapped in infinite loops.

As another achievement of our approach, we may refer to the solver-based algorithm for test data generation from UML state machine and using symbolic execution and constraint programming system. Different kinds of problems, e.g., linear, non-linear, and mixed integer non-linear, as well as satisfiability modulo theories can be solved using the proposed approach facilitated by the capability of using different state of the art and powerful solvers.

In the proposed approach, we consider Boolean, Integer, and Real data types. Also, we use early detection of infeasible paths during the generation of abstract test cases and remove infeasible paths from searching space. Pruning search space leads to improving execution time of search algorithm.

Furthermore, we can mention the boundary test data generation that has higher fault detection rate. The test data at the boundaries of feasible set are more valuable than those within feasible set. On the other hand, the coverage criteria are used for evaluating the quality of generated test cases as well as the criteria for conducting the search algorithm. Coverage criteria are converted into test goals and the search algorithm tries to satisfy them. Also, the satisfaction rate of these test goals can be used for evaluating the quality of generated test cases (higher test goals satisfaction rate means higher test cases quality).

In Section 4, several evaluations are performed for evaluating our approach. MoBaTeG shows good results for all of the evaluation criteria. Also, we compared our tool with ParTeG. The superiority of our tool has been shown over ParTeG. But, it should be noted that ParTeG supports several kinds of coverage criteria that our tool does not support. For example, we can refer to "Modified Condition/Decision Coverage" that is focused on the isolated impact of each atomic expression on the whole condition value [7].

Finally, we can mention that, as a result of our study, MoBaTeG is attained as a free, open source, and Eclipse-based tool.

5.2. Future work

In this paper, two coverage criteria have been used for conducting search algorithms as well as evaluating the quality of generated test cases. It is better to consider more coverage criteria for test case generation, "Condition/Decision Coverage" and "All Transition Pairs" to name a few.

Also, we plan to consider other kinds of unit tests as the format of the output of generated test cases. Hence, users can define output format of test cases based on the programming language of implementation. The meta-model that is proposed for test case generation can be used for this purpose.

We considered only some parts of OCL operations and expressions. We plan to add other OCL operations and expressions to the proposed approach. For example, we can mention set expressions. Another idea is to consider some operations and functions of AMPL that do not exist in OCL. But, we must consider them during parsing constraints and abstract syntax tree generation. Furthermore, we plan to support "Enumerated" data type by using indexed set of AMPL.

Several evaluations are carried out for evaluating our approach using academic and industrial case studies. We plan to evaluate our approach on more complex problems from industry in the future.

Finally, we plan to consider parallel state machines as well as history states for modeling dynamic behavior of system.

5.2.1. Threats to validity

Threats to the proposed approach and its validity are divided into internal and external threats. An

internal threat is that for input models, the proposed approach uses UML class diagram and UML state machine annotated with OCL. While most of the modelers are familiar with UML and can generate UML models easily, it may be difficult for other people. To mitigate this threat, several sample models have been provided along with our tool. An external threat to this study is the problem of generalizing the results. To mitigate this threat, several academic and industrial case studies with different levels of complexity are carried out, but it may not be enough to argue that the proposed approach can be used for any type of software. Therefore, as mentioned in subsection 5.2, the approach must be evaluated with more complex problems from the industry.

Acknowledgements

We are very much grateful to Stephan Weißleder and Felix Kurth for their valuable comments and guidance.

References

1. Selic, B. "The pragmatics of model-driven development", *Software, IEEE*, **20**(5), pp. 19-25 (2003).
2. Sendall, S. and Kozaczynski, W. "Model transformation: The heart and soul of model-driven software development", *Software, IEEE*, **20**(5), pp. 42-45 (2003).
3. Korel, B. "Automated software test data generation", *Software Engineering, IEEE Transactions on*, **16**(8), pp. 870-879 (1990).
4. El-Far, I.K. and Whittaker, J.A. "Model-based software testing", *Encyclo-pedia of Software Engineering* (2001).
5. Utting, M., Pretschner, A. and Legeard, B. "A taxonomy of model-based testing approaches", *Software Testing, Verification and Reliability*, **22**(5), pp. 297-312 (2012).
6. 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 (Jan. 2014).
7. Weißleder, S. "Test models and coverage criteria for automatic model-based test generation with UML state machines", PhD Thesis, Humboldt University of Berlin (2010).
8. Kundu, D. and Samanta, D. "A novel approach to generate test cases from UML activity diagrams", *Journal of Object Technology*, **8**(3), pp. 65-83 (2009).
9. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L. and Guo-liang, Z. "Generating test cases from UML activity diagram based on gray-box method", In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pp. 284-291, IEEE (2004).
10. Weißleder, S. and Sokenou, D. "Automatic test case generation from UML models and OCL expressions", In *Software Engineering (Workshops)*, pp. 423-426 (2008).
11. Ali, S., Iqbal, M.Z., Arcuri, A. and 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, IEEE (2011).
12. Malburg, J. and Fraser, G. "Combining search-based and constraint-based testing", In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 436-439, IEEE Computer Society (2011).
13. Tillmann, N. and De Halleux, J. "Pex-white box test generation for. net", In *Tests and Proofs*, pp. 134-153, Springer (2008).
14. Krieger, M.P. and Knapp, A. "Executing underspecified OCL operation contracts with a sat solver", *Electronic Communications of the EASST*, **15** (2008). https://journal.ub.tu-berlin.de/eceasst/article/view/176
15. Eclipse Foundation, "EclEmma." http://www.eclemma.org/. [Online; accessed 10-December-2013].

Biographies

Amin Rezaee received his BSc from the University of Fasa, Fars, Iran, in 2011, and his MSc from the University of Isfahan, Isfahan, Iran, in 2014, both in Computer Engineering (Software).

He was software testing researcher of International System Engineering & Automation Company (Irisa) in 2012. Amin Rezaee joined the Automation Systems Group of Irisa in 2014 as a software designer and programmer.

Bahman Zamani received his BSc from the University of Isfahan, Isfahan, Iran, in 1991, and his MSc from Sharif University of Technology, Tehran, Iran, in 1997, both in Computer Engineering (Software). He obtained his PhD degree in Computer Science from Concordia University, Montreal, QC, Canada, in 2009.

From 1998 to 2003, he was a researcher and faculty member of the Iranian Research Organization for Science and Technology (IROST), Isfahan branch. Dr. Zamani joined the Department of Computer Engineering at the University of Isfahan in 2009 as an Assistant Professor. His main research interest is Model-Driven Software Engineering (MDSE). He is the founder and director of MDSE research group at University of Isfahan.