Test Sequence Generation from Classification Trees

Peter M. Kruse
Berner & Mattner Systemtechnik GmbH

Berlin, Germany
peter.kruse@berner-mattner.com

Joachim Wegener
Berner & Mattner Systemtechnik GmbH

Berlin, Germany
joachim.wegener@berner-mattner.com

Abstract—**The classification tree methods allows for the automated test suite generation for given test objects. The important generation of test sequence can only be done manually. In this paper, we present an approach to use classification trees for test sequence generation. We will use existing approaches, e.g. from the field of researches on state machines. First results from our work are presented in this paper, including new dependency and generation rules.**

Keywords-testing; combinatorial; automatic test generation;

I. Introduction

The classification tree methods [1], together with some tool support [2], allows for the automated test suite generation for given test objects. It allows the generation of simple test cases only, which do not consider sequences of test inputs. Nevertheless, many applications require test cases which are composed by many sequential steps, so called test sequences. The important generation of test sequence can only be done manually. In this paper, we present an approach to use classification trees for test sequence generation. First results from our work are presented in this paper.

II. State of the Art

Classification tree method is introduced in [1]. The method bases on the category partition method introduced in [3].

The classification tree method describes a systematic approach to test case design. It consists of three steps: the definition of the system under test, the definition of constraints and the selection of test cases. In the first phase, all aspects of interests and their disjoint values are identified. Aspects of interests, also known as parameters, are called *classifications*; their corresponding parameter values are called *classes*.

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

Additionally, all classes can be refined using new classifications with classes again. This way, these classifications can be limited to the classes, they belong to. If the user adds the classification *c1* to the class *a*, then the classes of *c1* can only be selected, if *a* is selected. If the user selects a sibling-class of *a*, e.g. *b*, then the classification *c1* does not apply, its classes can not be selected.

These limitations are called *implicit dependencies.*

In a second step, *explicit dependencies* can be added to the classification tree. For any test object, there might be invalid parameters combinations. The classification tree method allows specifying these constraints. For example, in the classification tree *t*, if the class *a1* from classification *A* is selected, it can be required that class *b3* from classification *B* must be selected as well. The classification tree method allows to specify any kind of logical dependency rules, containing {*AND, OR, NOT,* $\Rightarrow$, $\Leftrightarrow$, *XOR, NOR, NAND*}.

The third step is the definition of test cases. The user needs to select one class per each classification in the classification tree without violating the existing dependency rules.

Tool support for the classification tree method is introduced in [2]. An editor is presented allowing automatic test case generation instead of manual test case definition. The tool presented allows creating test suites that fulfill certain coverage levels. The supported coverage levels are *minimal* combination, *pairwise* combination, *threewise* combination, and *complete* combination.

III. Enhancement

Current tools supporting the classification tree method are capable of generating test suites. The generation uses the classification tree itself, dependency rules and generation rules. The classification tree describes parameters and parameter values of the system under test. Dependency rules contain constraints between parameter values. They apply per test case. Within each test case, no dependency rule must be violated. The generation rules describe the desired coverage level for the resulting test suite. The test suite as a whole must respect the generation rule.

For our new approach, we want to enable test sequence generation from classification trees. Analogous to existing approaches, we identify three kinds of parameters for test sequence generation, the classification tree itself, dependency rules and generation rules. Again, the classification tree holds all parameters and their corresponding values of the system under test. For dependency rules, we extend existing rules to new rules describing constraints between single test steps. Our new rules apply per test sequence. Within each test sequence,

This work is supported by EU grant ICT-257574 (FITTEST).

no dependency rule must be violated. The generation rules describe desired coverage levels for the resulting set of test sequences. The set of test sequences as a whole must respect the generation rule.

A. New Dependency Rules

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

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

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

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

B. New Generation Rules

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

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

Some of the new generation rules are similar to the original generation rules, e.g. the desired coverage level or set of all parameter-value combination.

C. Approach

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

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

This obviously leads to numerous sequences.

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

IV. Implementation

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

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

For generation rules, we used *computational tree logic* (CTL). CTL rules are not valid per single sequence but refer to the hole set of generated test sequences.

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

The set of valid states is the set of all valid test cases. This already can be very large. Therefore, the user starts with the generation of classical test cases and can use these as a subset of all valid test cases for the proceeding test sequence generation.

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

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

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

V. Related Work

Related work falls into two parts, combinatorial and pairwise testing on the one hand. On the other side, there is a lot of fundamental work on test sequence generation and validation.

A. *Combinatorial Testing*

A good overview on combinatorial testing and combinatorial test case generation is given by Grindal et al. in [4] and by Kuhn et al. in [5]. A good survey that focuses on combinatorial testing with constraints is given in [6].

Kuhn et al. present several different empirical studies on the relation between fault detection and level of parameter interaction based on error reports of different systems [7], [8], [9]. They are resulting that a pairwise coverage had led to a fault-coverage of 70% to 97%. A 6-wise combination had led to a fault-coverage of 100% on all software systems under research. They hypothesize that there is a $t < n$, so that the t-wise combination detects all faults. His restriction is, to support his hypothesis, he needs to evaluate more software systems with empirical studies.

Burr and Young do research on branch coverage by pairwise combination of input parameters [10]. The system under test is a tool for email conversion. They gain 97% branch coverage with less than 100 test cases. A complete test would require 27 trillion test cases.

Since finding a minimal test set for any given coverage criterion is known NP-complete, all approaches presented are either approximations or heuristics. They try to find a test suite as small as possible.

Prominent tools are AETG, PICT and CASA, to name just a few: AETG has been developed at Bellcore and is presented by Cohen et al. [11]. AETG uses a deterministic greedy-algorithm. PICT [12] is a combinatorial test case generation tool. It has been used at Microsoft since 2000. A general constraint handling approach is presented in [6]. Their algorithm is called covering arrays by simulated annealing (CASA).

B. *Test Sequence Generation and Validation*

Model checking aims to prove certain properties of the execution of a program by completely analyzing its finite state model algorithmically [13], [14]. When these mathematically defined properties hold for all possible states of the model, then it is proven that the model satisfies the properties. However, when this property is violated somewhere, the model checker tries to provide a counterexample which is the sequence of states that leads to the situation which violates the property. A big problem with model checking is the state explosion problem, where the number of states can grow very quickly when the program becomes more complex, increasing the total number of possible interactions and/or values. Therefore, an important part of research on model checking is state space reduction, to minimize the time required to traverse the entire state space. The Partial-Order Reduction (POR) method is regarded as a successful method for reducing this state space [14]. Other methods in use are *symbolic model checking*, where the construction of a very large state space is avoided by use of equivalent formulas in propositional logic, and *bounded model checking*, where the construction of the state space is limited to a fixed number of steps.

Two temporal logics are compared and debated extensively [15], *Linear temporal logic* (LTL) and *Computation Tree Logic* (CTL).

Heimdahl et al. [16] briefly surveys a number of approaches in which test sequences are being generated using model checking techniques. The common idea is to use the counter-example generation feature of model checkers to produce relevant test sequences.

Krupp and Mueller [17] introduce an interesting application of CCTL logic for the verification of classification tree test sequences. Using a real-time model checker, the test sequences and their transitions are being verified using a combination of I/O interval descriptions and CCTL expressions.

Several researchers propose other approaches for test sequence generation. Wimmel et al. [18] propose a method of generating test sequences using propositional logic. Ural [19] describes four formal methods for generating test sequences based on a finite-state machine (FSM) description. The question to be answered by these test sequences is whether or not a given system implementation conforms to the FSM model of this system. Test sequences consisting of inputs and their expected outputs are derived from the FSM model of the system, after which the inputs can be fed to the real system implementation. Finally, the outputs of the model and the implementation are compared.

Bernard et al. [20] have done an extensive case study on test case generation using a formal specification language called B. Using this machine modeling language, a partial model of the GSM 11-11 specification has been built. A system of equivalent constraints is then derived from this specification, after which a constraint solver is used to calculate boundary states and test cases.

Binder [21] lists a number of different oracle patterns that can be used for software testing, one of which is the simulation oracle pattern. With this pattern a simulation of the system is done using a simplified version of the system implementation, after which the results of the simulation are compared to the results of the real system. We can regard the formal model of the system as the simulation of the system, from which expected results are derived.

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

VI. Conclusion

In this paper, we presented an approach for automatic test sequence generation from classification trees. We defined new dependency rules and generation rules. The new dependency rules are a superset of existing dependency rules, extended by expressions for describing constraints between several test steps. The new generation rules contain parameters for the definition of test amount. They can specify the coverage levels and and number of repetitions. This allows for the systematic generation of test sequences for classification trees.

Finishing the prototype implementation of the test sequence generation in CTE XL Professional, future work will focus on the evaluation of the presented approach, particularly with regard to scalability and performance of the proposed solution.

REFERENCES

[1] M. Grochtmann and K. Grimm, "Classification trees for partition testing," Softw. Test., Verif. Reliab., vol. 3, no. 2, pp. 63–82, 1993.

[2] E. Lehmann and J. Wegener, "Test case design by means of the CTE XL," Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000), Kopenhagen, Denmark, December, 2000.

[3] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," Communications of the ACM, vol. 31, no. 6, pp. 676–686, 1988.

[4] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," Softw. Test., Verif. Reliab., vol. 15, no. 3, pp. 167–199, 2005.

[5] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," IT Professional, vol. 10, no. 3, pp. 19–23, 2008.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis. New York, NY, USA: ACM, 2007, pp. 129–139.

[7] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," IEEE Transactions on Software Engineering, vol. 30, pp. 418–421, 2004.

[8] D. Wallace and D. Kuhn, "Failure modes in medical device software: an analysis of 15 years of recall data," INTERNATIONAL JOURNAL OF RELIABILITY QUALITY AND SAFETY ENGINEERING, vol. 8, pp. 351–372, 2001.

[9] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE, 2002, pp. 91–95.

[10] K. Burr and W. Young, "Combinatorial test techniques: Tablebased automation, test generation and code coverage," in Proc. of the Intl. Conf. on Software Testing Analysis & Review. Citeseer, 1998.

[11] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," IEEE Transactions on Software Engineering, vol. 23, pp. 437–444, 1997.

[12] J. Czerwonka, "Pairwise testing in real world, practical extensions to test case generators," in Proceedings of 24th Pacific Northwest Software Quality Conference. Citeseer, 2006, pp. 419–430.

[13] D. Boˇsnaˇcki and S. Edelkamp, "Model checking software: on some new waves and some evergreens," Int. J. Softw. Tools Technol. Transf., vol. 12, pp. 89–95, May 2010.

[14] R. Jhala and R. Majumdar, "Software model checking," ACM Comput. Surv., vol. 41, pp. 21:1–21:54, October 2009.

[15] M. Y. Vardi, "Branching vs. linear time: Final showdown," in Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ser. TACAS 2001, 2001, pp. 1–22.

[16] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-generating test sequences using model checkers: A case study," in 3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003), 2003.

[17] A. Krupp and W. Müller, "Modelchecking von klassifikationsbaum-testsequenzen," 1 Apr. 2005, gI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", München.

[18] G. Wimmel, H. Loetzbeyer, A. Pretschner, and O. Slotosch, "Specification based test sequence generation with propositional logic," 2000.

[19] H. Ural, "Formal methods for test sequence generation," Comput. Commun., vol. 15, pp. 311–325, June 1992.

[20] E. Bernard, B. Legeard, X. Luck, and F. Peureux, "Generation of test sequences from formal specifications: Gsm 11-11 standard case study," Softw. Pract. Exper., vol. 34, pp. 915- 948, August 2004.

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