SOFTWARE TESTING METHODS USING FORMAL MODELS

PROF. NIRVIKAR KATIYAR, DR. RAGHURAJ SINGH
HOD CS/IT AXIS COLLEGES Kanpur
HOD CS/IT HBTI Kanpur

ABSTRACT

Model-based testing relies on models of a system under test and/or its environment to derive test cases for the System. Model-based testing refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases. Therefore, the key points of model-based testing are the modeling principles for test generation, the test generation strategies and techniques, and the concretization of abstract tests into concrete, executable tests. Model-based functional testing is focused on comparing the system under test to a test model. This comparison usually consists of automatically generating a test suite from the test model, executing the test suite, and comparing the observable behavior to the expected one. Important advantages of model-based testing are formal test specifications that are close to requirements, traceability of these requirements to test cases, and the automation of test case design, which helps reducing test costs. Testing cannot be complete in many cases: For test models that describe, e.g., non terminating systems, it is possible to derive a huge and possibly infinite number of different test cases. Coverage criteria are a popular heuristic means to measure the fault detection capability of test suites. They are also used to steer and stop the test generation process.

KEYWORDS: *Uml, Ocl, Sorting Machine, Schumacher Freight Elevator, Location Analyzer, Boundary Value Analysis, Triangle Categorization*

INTRODUCTION

Model-based testing (MBT) is an increasingly widely-used technique for automating the generation and execution of tests. There are several reasons for the growing interest in using model-based testing:

  The complexity of software applications continues to increase , and the user’s aversion to software defects is greater than ever, so our functional testing has to become more and more effective at detecting bugs;

  The cost and time of testing is already a major proportion of many projects (sometimes exceeding the costs of development), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically as well as executing them automatically.

  The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good ROI;
  Model-based testing renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation. Testing is one of the most important means to validate the correctness of systems. The costs of testing are put at 50% [1, 2 and 3] of the overall project costs. There are many efforts to decrease the costs for testing, e.g. by introducing automation. There are many different testing techniques, processes, scopes, and targets. Functional testing consists of comparing the system under test (SUT) to a specification. A functional test detects a failure if the observed and the specified behavior of the SUT differ. Model-based testing is about using models as specifications. Several modeling languages have been applied to create test models, as [4 and 5], the Unified Modeling Language (UML) [6], or the Object Constraint Language (OCL) [7]. Model-based testing allows deriving test suites automatically from formal test models. This paper is focused on automatic model-based test generation with UML state machines and OCL expressions.

1.1 UNIFIED MODELING LANGUAGE (UML): UML is similar to finite state machines, except for the fact that it is used to describe complicated behaviors of software, thus the simple graphical representation of a state machine is replaced by a structured language.

1.2 OBJECT CONSTRAINT LANGUAGE (OCL): OCL is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and is a highly expressive language. The language allows modelers to write constraints at various levels of abstraction and for various types of models. It can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post condition of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) (which is a standard defined by Object Management Group (OMG) for defining meta-models). This subset of OCL has been largely used in the definition of UML for constraining various elements of the language.

Although testing with state charts, state diagrams, or state machines has been investigated for several decades, there are still many unexplored aspects and issues left to be solved. Testing is often incomplete, i.e. cannot cover all possible system behaviors. There are several heuristic means to measure the quality of test suites, e.g. fault detection, mutation analysis, or coverage criteria. These means of quality measurement can also be used to decide when to stop testing.

SELECTION OF INPUT VALUES CRITERIA

This paper contains a motivation for our new testing approach: We shortly describe the benefits of combining different kinds of coverage criteria. For the description of reactive systems, graph-based models like UML state machines are often applied. As described above, the quality or the adequacy of test cases is often measured with coverage criteria that are focused on data flow, control flow, or transition sequences [8].. One important problem of the existing test generation approaches is the selection of concrete input values: The approaches often use random input values and determine whether the specified model elements are covered. There are approaches that allow the user to define input values manually [9]. Such approaches mostly do not include information about the quality of the input values. There is need to apply means of representative input value selection. An alternative approach to user-defined or manually chosen input values is partition testing, which divides the input space into input partitions. Each input partition is a set of constraints for objects that are assumed to trigger the same system behavior. This assumption is called the uniformity hypothesis [10]. As a consequence of this assumption, only a few representatives of these equivalence classes are chosen instead of all values. Experience shows that many faults occur near the boundaries of equivalence classes [11 and 12]. There are corresponding boundary-based coverage criteria for the selection of representatives from input partitions [13]. These coverage criteria can be combined with other approaches from partition value analysis like additionally selecting random elements from the inside of partitions [14]. The issue of input partitions is that they are only applied to the input space of non-reactive systems [15], i.e. systems that behave like a function: Only the input space is considered and information about state changes or further behavior is neglected. Both approaches of creating abstract test cases and selecting concrete input values are important aspects of automatic model-based test generation. However, they are often considered in isolation because each of them is only applied to reactive or non-reactive systems, respectively. The core idea of our test generation approach is to combine both approaches and make them applicable to reactive systems.

FORMAL SPECIFICATIONS

In this paper we present five test models that are used for automatic test generation: The sorting machine, freight elevator are artificial examples that, despite their small size, partly describe behavior that is common in practice. The triangle classification is a standard example for challenging test input data generation problems [8]. The test model for a track control is an academic test model that is part of the UnITeD project [16] of the University Erlangen. The train control test model is part of our industrial cooperation project with the German supplier of railway signaling solutions. All state machines describe the behavior of the class that is the context of the state machine and defines all structural elements. All test models contain attributes of linear-ordered types. Boundary value analysis is an important task for such applications. The corresponding test cases have to contain values that check even small violations of the derived input parameter boundary values. Our prototype implementation ParTeG [Weib] has been used to generate test suites for all these test models. This also includes boundary value analysis. The prototype is used in further industrial applications. However, we considered number of presented test models sufficient for this paper. The selection of test models is by no means a restriction of ParTeG's application fields, e.g., ParTeG is one of special value for train-related applications. That depends on exact values and boundaries of linear-ordered value types.

SORTING MACHINE

The sorting machine test model describes the behavior of a machine that wraps up an object and subsequently sorts the resulting package depending on its size. The object is put into a plastic box filled with foam. The package size depends on the size of the object. A possible application field of such machines can be found in post offices. Figure-1shows a state machine and Figure-2 shows the corresponding class diagram of such a sorting machine. The diagrams define the rules for the packaging as follows: Due to wrapping up the objects, the original width of the object should be doubled by foam plus two extra size units for each side of a plastic box: (width = (object.width + 2) * 2). The height is handled a bit different, and the corresponding equation is: (height = (object.height * 2) + 2). Our sorting machine's task is to sort incoming items depending on the size after their wrapping so that they fit into given transport containers. The operations of the class Sorting Machine contain post conditions and are referenced from the state machine. The post conditions of these operations influence the behavior of the state machine: The size of the package is described in the post condition of the operation detect Item.

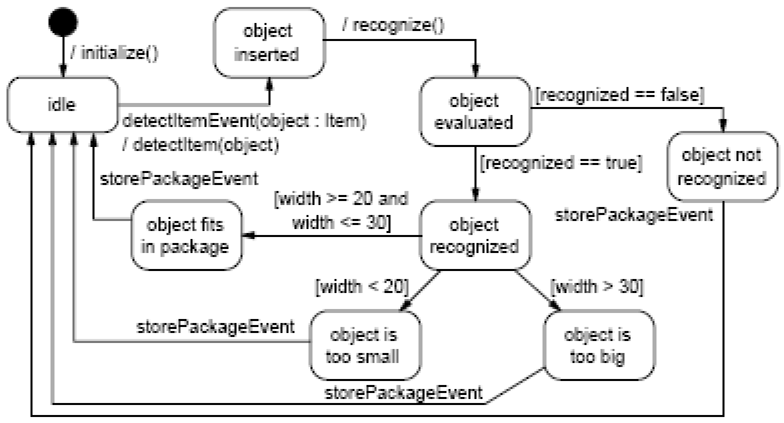

Figure-1 State machine for a sorting machine

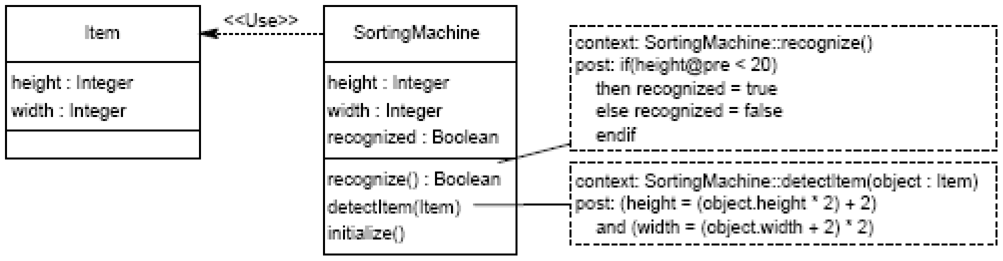

Figure-2 Class diagram of a sorting machine

The sorting is determined by the post condition and recognize by the guard conditions. The outgoing transitions of the states object recognized and object evaluated: Objects that are too tall are sorted out using the guards of the outgoing transitions of state object evaluated. These objects are considered as not recognized. The remaining objects are sorted depending on their width. The values of height and width of the parameter of detect item, both influences the packaging. The only exceptions are tall objects, for which the width is unimportant.

SCHUMACHER FREIGHT ELEVATOR

The Schumacher freight elevator behaves similar to a normal elevator. It can carry all weights up to a maximum weight, and the user can press a button to select the target floor of the elevator. Additional features are that the elevator can move faster if it is empty and that certain floors require a certain minimum user rank or authority. Schumacher freight elevators are built tough and dependable to stand up against heavy everyday use for years. Schumacher Elevator has the ability to produce your standard elevator package needs, while specializing in custom engineering with a complete design staff.
Figure-3 shows the state machine that describes the behavior of a freight elevator. If the elevator is in state idle, the user of the elevator can insert and remove weight. The user can also press a button for the elevator's next target floor.

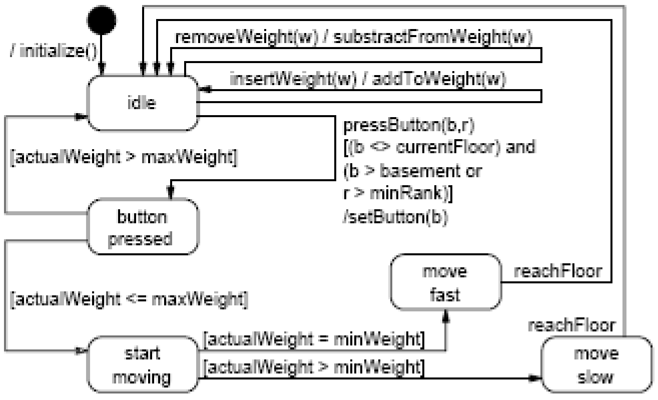
/ initialize()
removeWeight(w) / substractFromWeight(w)
insertWeight(w) / addToWeight(w)
idle
pressButton(b,r)
[(b <> currentFloor) and
(b > basement or
r > minRank)]
/setButton(b)
[actualWeight > maxWeight]
button pressed
[actualWeight <= maxWeight]
move fast
reachFloor
reachFloor
start moving
[actualWeight = minWeight]
[actualWeight > minWeight]
move slow

Figure-3 State machine for the Schumacher freight elevator control

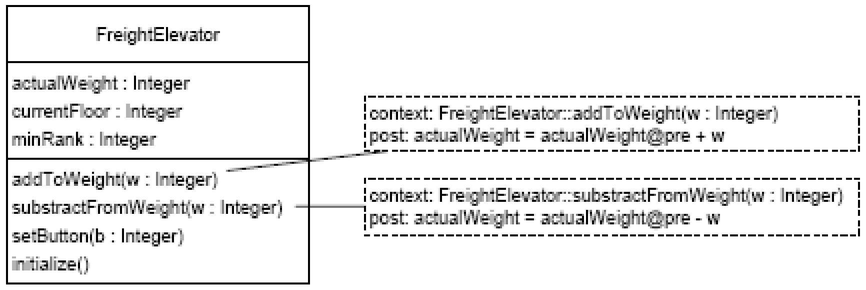
FreightElevator
actualWeight : Integer
currentFloor : Integer
minRank : Integer
addToWeight(w : Integer)
substractFromWeight(w : Integer)
setButton(b : Integer)
initialize()
context: FreightElevator::addToWeight(w : Integer)
post: actualWeight = actualWeight@pre + w
context: FreightElevator::substractFromWeight(w : Integer)
post: actualWeight = actualWeight@pre - w

Figure-4 Class diagram of the freight elevator control

As a reaction, the elevator checks the user's rank and whether the current weight exceeds the maximum weight. Subsequently, the elevator begins to move at an appropriate velocity until the target floor is reached. Like in the previous example, the post conditions of operations in the class diagram influence the behavior of the state machine. (Refer to Figure-4)

TRIANGLE CATEGORIZATION

The triangle categorization example is a standard example in testing literature [1 and 8]. The task is to classify triangles whose three side lengths are described by three integer values. The triangle is invalid if one side length is less or equal to zero or if the sum of two side lengths is less or equal to the third side length. Valid triangles can be scalene, isosceles, or equilateral (see Figure-5). The issue of this example is the huge number of possible parameter value combinations.

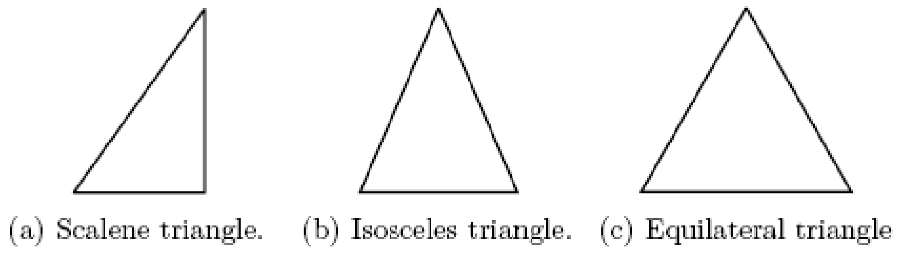

Figure- 5 valid triangle categorization results

For this example, boundary value analysis is a good means to select proper input values. Since the conditions for the triangle classification include many mutual dependencies of the input parameters, there are many possible boundary values. This results in many correspondingly necessary test cases. There are many ways to describe the triangle categorization problem. As define OCL expressions to classify triangles [8]. We use UML state machines.

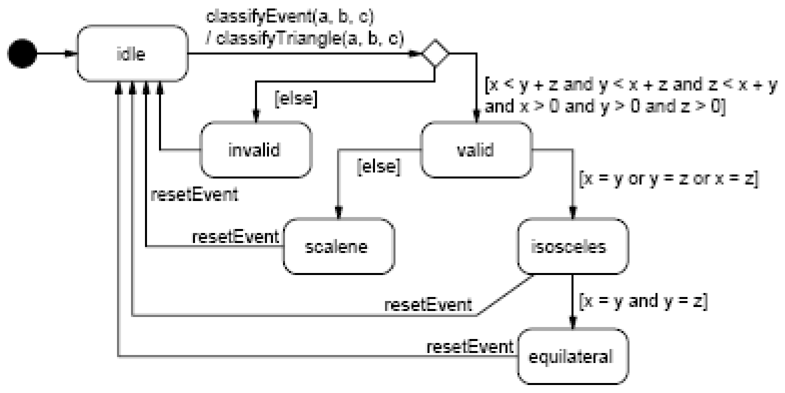

Figure-6 State machine for triangle categorization

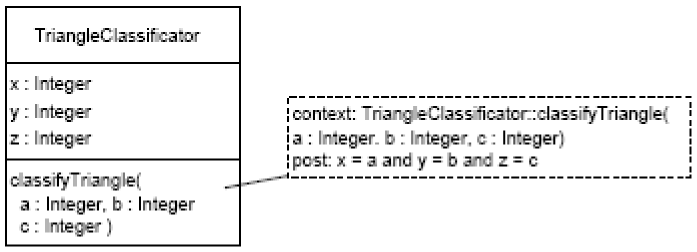
TriangleClassificator
x : Integer
y : Integer
z : Integer
classifyTriangle(
a : Integer, b : Integer
c : Integer)
context: TriangleClassificator::classifyTriangle(
a : Integer. b : Integer, c : Integer)
post: x = a and y = b and z = c

Figure-7 Class diagram for triangle categorization

Figure-6 shows such a state machine. The corresponding class is shown in Figure-8. The post condition of the operation classify Triangle (a, b, c) represents a simple mapping from input parameters a, b, and c to system attributes x, y, and z.

ROUTE MONITORING

The track control example is part of the UnITeD project [16] of the University Erlangen. This example describes the control of a railway track and includes the steering of trains depending on their priority, train number and the track status. We present a part of the test model to show the frequent use of linear-ordered types. We never use the complete model for test generation.

Figure-9 depicts a part of the state machine and the corresponding class diagram. Note that there are several integer attributes for which boundary value analysis is a proper means to improve the test results.

i). BOUNDARY VALUE ANALYSIS

- Boundary value analysis (BVA) is based on testing at the boundaries between partitions.

- Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).

- As an example, consider a printer that has an input option of the number of copies to be made, from 1 to 99. To apply boundary value analysis, we will take the minimum and maximum (boundary) values from the valid partition (1 and 99 in this case) together with the first or last value in respectively in each of value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case).

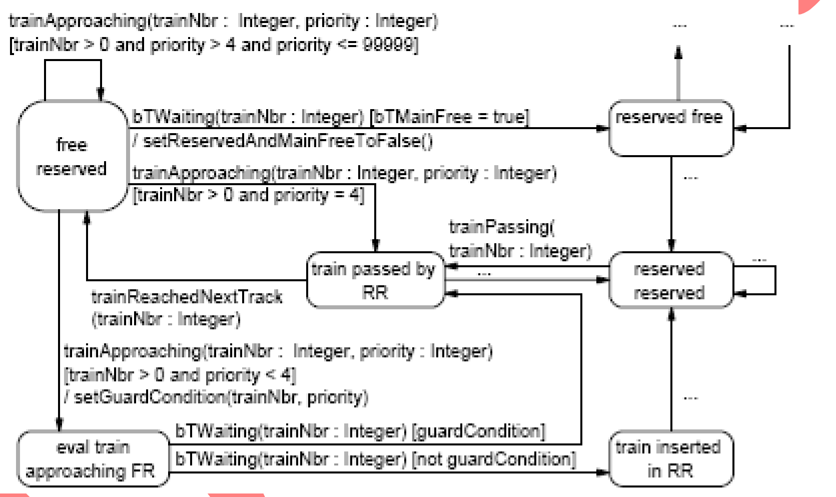
trainApproaching(trainNbr : Integer, priority : Integer)
[trainNbr > 0 and priority > 4 and priority <= 99999]
...
...
free reserved
bTWaiting(trainNbr : Integer) [bTMainFree = true]
/ setReservedAndMainFreeToFalse()
reserved free
trainApproaching(trainNbr : Integer, priority : Integer)
[trainNbr > 0 and priority = 4]
...
trainPassing(
trainNbr : Integer)
train passed by RR
...
reserved reserved
...
trainReachedNextTrack
(trainNbr : Integer)
trainApproaching(trainNbr : Integer, priority : Integer)
[trainNbr > 0 and priority < 4]
/ setGuardCondition(trainNbr, priority)
...
eval train approaching FR
bTWaiting(trainNbr : Integer) [guardCondition]
bTWaiting(trainNbr : Integer) [not guardCondition]
train inserted in RR

Figure-8 State machine of the route monitoring

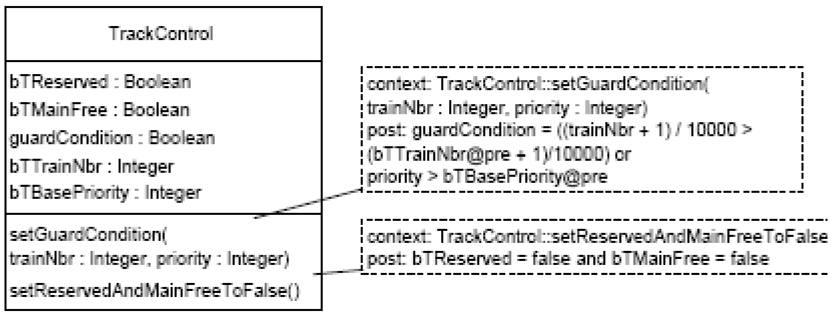
TrackControl
bTReserved : Boolean
bTMainFree : Boolean
guardCondition : Boolean
bTTrainNbr : Integer
bTBasePriority : Integer
setGuardCondition(
trainNbr : Integer, priority : Integer)
setReservedAndMainFreeToFalse()
context: TrackControl::setGuardCondition(
trainNbr : Integer, priority : Integer)
post: guardCondition = ((trainNbr + 1) / 10000 >
(bTTrainNbr@pre + 1)/10000) or
priority > bTBasePriority@pre
context: TrackControl::setReservedAndMainFreeToFalse
post: bTReserved = false and bTMainFree = false

Figure-9 Class diagram of the route monitoring

In this example we would have three equivalence partitioning tests (one from each of the three partitions) and four boundary value tests. Consider the bank system described in the previous section in equivalence partitioning.

Example:
A store in city offers different discounts depending on the purchases made by the individual. In order to test the software that calculates the discounts, we can identify the ranges of purchase values that earn the different discounts. For example, if a purchase is in the range of $1 up to $50 has no discounts, a purchase over $50 and up to $200 has a 5% discount, and purchases of $201 and up to $500 have a 10% discounts, and purchases of $501 and above have a 15% discounts.

LOCATION ANALYZER

The Motorola location analyzer is an application used to help determine an optimal location to setup a mesh enabled architecture wireless network infrastructure device. The location analyzer application takes into account various parameters and the application then determines a deployment quality number that varies between 0 and 100. The parameters used for calculation are collected from a wireless modem card (WMC6300) the application operates on an analysis run principal. An analysis run is a finite duration of time during which date is gathered from awmc6300 and deployment. As a typical use the user will start an analysis run and save several snapshots for later comparison. If required the user can then move to another location and repeat this process. After you have collected analyzed data from several locations, you can compare snapshots history data and log file data to determine the best location from those that were analyzed.
The test model for a location analyzer is part of our industrial cooperation with the German supplier of railway signaling solutions. The test model describes the communication behavior between modules of a train and the track. The task of these modules is to determine the train’s position. Since safety is an important issue for this company, the model describes several cases for emergency situations. Transitions are all triggered by call events. All generated tests are functional tests without time information. In order to protect the intellectual property of the company, we only provide an anonymised version of the test model. Due to the test model size and complexity, Figure-10 depicts only a part of the state machine. Since the model is anonymised, there is no sense in presenting the class diagram. Again, we use the complete model for test generation.

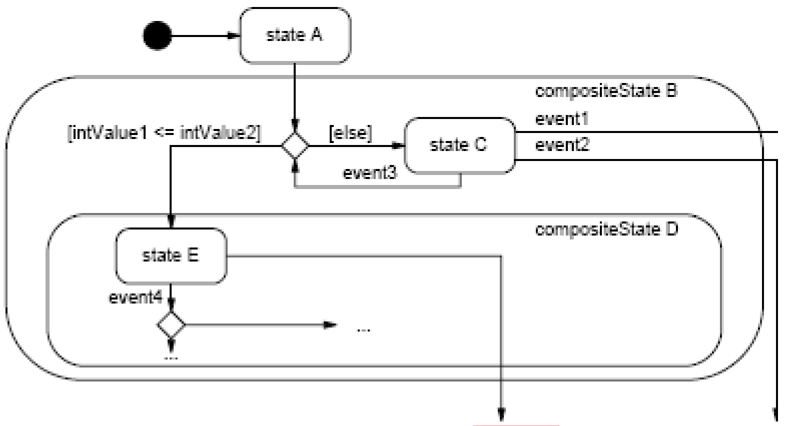

Figure-10 Anonymised part of the train control state machine

CONCLUSION

In this paper, The comparison made between model based testing and traditional, manual testing showed clearly that the former is the cheaper and faster method, and almost as effective in terms of the code coverage reached as the latter one. Model based testing was a little bit behind the traditional method in code coverage we presented a new approach to test suite generation, which is focused on combining data-flow-based, control-flow-based, or transition based coverage criteria with boundary-based coverage criteria. We also introduced example test models from literature, academia, and industry, and we used them to evaluate the impact of combined coverage criteria on the test suites generated by our prototype implementation ParTeG. As a major result of the corresponding mutation analysis, the satisfaction of combined coverage criteria results in a higher fault detection capability of the generated test suite than the satisfaction of single coverage criteria. Since the latter is the state of the art, we also pointed out our contribution and showed the advantages of combining coverage criteria. Small comparisons with test suites generated by commercial tools further accomplish the advantages of ParTeG over the state of the art in automatic model-based testing approach.

REFERENCES

1. Myers, Glenford J.: Art of Software Testing. John Wiley & Sons, Inc., New York, NY, USA, 1979. ISBN 0471043281.
2. Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc: Testing Computer Software, 2nd Ed. John Wiley and Sons, Inc., New York, USA, 1999. ISBN 0-471-35846-0.
3. Sommerville, Ian: Software Engineering. Addison-Wesley, New York, USA, 2001.
4. Abrial, Jean-Raymond: Formal Methods: Theory Becoming Practice. In: Journal of Universal Computer Science, volume 13(5):pp. 619–628, 2007.
5. Spivey, Mike: The Z Notation: A Reference Manual. Prentice- Hall International Series in Computer Science, 1992. ISBN 0139785299.

6. Object Management Group: Unified Modeling Language (UML), version 2.1. http://www.uml.org, 2007.
7. Object Management Group: Object Constraint Language (OCL), version 2.0. http://www.uml.org, 2005.
8. Utting, Mark; Legeard, Bruno: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123725011.
9. Budnik, Christof J.; Subramanyan, Rajesh; Vieira, Marlon: Peer-to-Peer Comparison of Model-Based Test Tools. In: Hegering, Heinz-Gerd; Lehmann, Axel; Ohlbach, Hans Jürgen; Scheideler, Christian, editors, GI Jahrestagung (1), volume 133 of Lecture Notes in Informatics, pp. 223–226. GI, 2008. ISBN 978-3-88579-227-7.
10. Bernot, Gilles; Gaudel, Marie Claude; Marre, Bruno: Software Testing Based on Formal Specifications: A Theory and a Tool. In: Software Engineering Journal, volume 6(6):pp. 387– 405, 1991. ISSN 0268-6961.
11. White, Lee J.; Cohen, Edward I.: A Domain Strategy for Computer Program Testing. In: IEEE Transactions on Software Engineering, volume 6(3):pp. 247–257, 1980. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.1980.234486.
12. Clarke, Lori A.; Hassell, Johnette; Richardson, Debra J.: A Close Look at Domain Testing. In: IEEE Transactions on Software Engineering, volume 8(4):pp. 380–390, 1982. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10. 1109/TSE.1982.235572.
13. Kosmatov, Nikolai; Legeard, Bruno; Peureux, Fabien; Utting, Mark: Boundary Coverage Criteria for Test Generation from Formal Models. In: ISSRE'04: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 139–150. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2215-7. doi:http://dx.doi.org/10.1109/ ISSRE.2004.12.
14. Binder, Robert V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9.
15. Broy, Manfred; Jonsson, Bengt; Katoen, Joost P.: Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer, August 2005. ISBN 3540262784. doi:http://dx.doi.org/http://dx.doi.org/10.1007/ b137241.
16. Pinte, Florin; Saglietti, Francesca: UnITeD-Unterstützung Inkrementeller TestDaten. http://www11.informatik.unierlangen. de/Forschung/Projekte/United/ index.html, 2007.