Well-Defined Coverage Metrics for the Glass Box Test

Rainer Schmidberger

Well-defined Coverage Metrics for the Glass Box Test

Rainer Schmidberger

ISTE (Institute for Software Technology), Stuttgart University, Germany
rainer.schmidberger@informatik.uni-stuttgart.de

Abstract. The Glass Box Test (GBT), also known as White Box Test or Structural Test, shows which parts of the program under test have, or have not, been executed. Many GBT tools are available for almost any programming language. Industry standards for safety-critical software require a very high or even complete coverage. At first glance, the GBT seems to be a well-established and mature testing technique that is based on standardized metrics. But on closer inspection, there are several serious shortcomings of the underlying models and metrics which lead to very imprecise, inconsistent coverage results of the various GBT tools. In this paper, a new and precise model for the GBT is presented. This model is used as a reference for the precise definition of all the popular coverage metrics that are around. The tool CodeCover which was developed in the University of Stuttgart is an implementation that strictly follows those definitions.

Keywords: Glass Box Test, White Box Test, Structural Test, coverage testing, coverage tools

1 Introduction

In industrial practice testing is the technique most widely used to find errors in programs or to demonstrate a certain quality of a program. Accordingly, companies spend a significant part of their project budgets on testing, which in many cases does not achieve the desired quality – (too) many serious defects remain undetected. As a starting point for the improvement of the test, the Glass Box Test (GBT) can be used, also known as White Box Test or Structural Test, that shows which parts of the program under test have, or have not, been executed. This degree of execution is called coverage. It can be used as a test completion criterion or as input for developing test cases [1, 3, 8, 14, 18]. Many GBT tools are available for almost any programming language. Industry standards for safety-critical software require a very high or even complete coverage (e.g. [8, 12]). Empirical studies clearly indicate that higher GBT coverage correlates with lower post-release defect density [2, 4, 13].

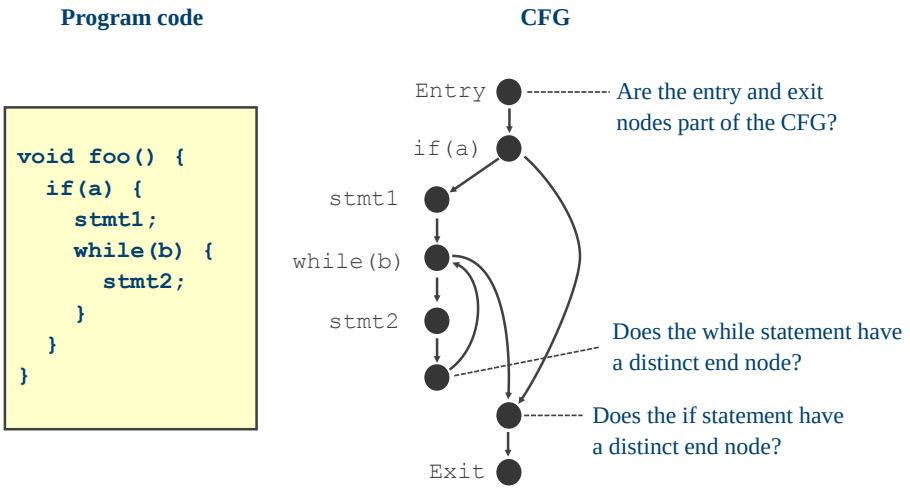
At first glance, the GBT seems to be a well-established and mature testing technique, but it is worth having a closer look at the underlying models and metrics.

adfa, p. 1, 2011.

For the GBT, typically the control flow graph (CFG) is used to build an abstraction model of the original program code. Most popular GBT metrics are defined with respect to the CFG [1, 3, 10, 11]. The statement coverage of a program execution, for example, is defined as the node coverage of the program's corresponding CFG. The branch coverage is defined as the edge coverage.

Thus, while the GBT metrics based on the CFG are precisely defined, the transformation of the real programs into the model is imprecise and ambiguous. For example, it is undefined whether the CFG representation of an if- or while-statement has a distinct end node which corresponds to "EndIf" or "EndWhile". **Fig. 1** shows an example. The classification of nodes as entry or exit nodes is ambiguous as well: Some authors add a distinct entry and exit node to the CFG, others do not.

Fig. 1. Ambiguities in CFG modelling



What is more severe than these details is the fact that there is no representation of exception handling in the CFG. But exception handling is an important part of modern programming languages. In addition, the CFG does not provide any means for handling the GBT-relevant expressions such as the conditional expression or the short-circuit operations of Boolean expressions [9]. As a result, it is not possible to model compound Boolean operations for handling logic-based coverage criteria like term coverage [19] or MC/DC [1, 19]. As a consequence of this unsatisfactory model and metric definition, the most important GBT tools for Java [17] show completely different coverage results (**Table 1**) for the same execution of a given 45-statement reference program [16]. Furthermore, those tools are not based on well-defined metrics. (What does line coverage or instruction coverage mean?)

While these differences can be explained partly by the different instrumentation techniques of the GBT tools, there is no "reference" which the tool developer could use as a specification and for comparing results.

Table 1. Coverage results for the same reference execution

	Statement coverage	Branch coverage
CodeCover Version: 1.0.2.2	62,8 %	Branch: 50,0 % Block: 52,2 %
Clover Version: 3.1.0	58,5 %	
Emma Version: v2.1.5320	Line: 62,0 %	Block: 54,0 %
EclEmma Version: 2.2.1	Instruction: 56,7 % Line: 63,6 %	Block: 50,0 %
eCobertura Version: 0.9.8	64,3 %	Branch: 50,0 %
CodePro Version: 7.1.0	Instruction: 57,7 % Line: 58,5 %	Block: 60,6 %
Rational Application Developer Version 9.0.0	Line: 67,0 %	

2 A new and precise model for the GBT

The GBT model described below is intended to provide such a reference. It should meet the following requirements:

- The model forms the basis on which both the popular control-flow based metrics as and the logic and conditional expression based metrics can be defined.
- The model supports exception handling and provides an easy and clear definition of how coverage metrics are applied to exception handling.
- There is an easy and precise transformation rule to transform real programs into the model.
- The model does not depend on any particular programming language. An algorithm implemented in different programming languages should have the same model representation.
- The model specifies how to place the probes that count the execution of the relevant code items in the instrumented program.

Such a GBT model is now presented in three steps:

1. Definition of a primitive language (RPR = Reduced Program Representation) which is reduced to the GBT-relevant aspects of the real programming languages. It provides control flow, exception handling, and expressions. These GBT-relevant aspects are mapped in so-called GBT items such as statements, statement blocks, and Boolean or conditional expressions. This abstraction has two goals: First, to make metric definition easier because the metrics are based on only a few distinct elements. And, second, to keep the model independent of a particular programming language.

2. Definition of the execution semantics using Petri nets. This part of the model precisely defines the control flow inside and between the GBT items for both normal and abrupt completion. The Petri net model specifies precisely how the original program has to be instrumented.

3. Definition of the coverage metrics based on the Petri nets.

2.1 The Reduced Program Representation (RPR)

RPR defines all GBT-relevant aspects of the original program code and suppresses all irrelevant attributes like numerical expressions or design elements such as inheritance, interface or visibility. RPR is subdivided into two parts: First, the control flow part which covers the typical control flow statements like decision, loop, and switch statement. In the second part the GBT-relevant expressions such as the conditional expression or Boolean terms are described. The control structures of RPR correspond to the principles of structured programming as described by Dahl, Dijkstra, and Hoare [18]. In addition, a try-statement covers the typical exception handling of modern languages. *Program* represents a method or procedure. The *StatementBlock* subsumes all blocks and branches like then or else blocks, method blocks, and catch blocks. *Statement* and *StatementBlock* are given a unique identifier that does not exist in the original program. This identifier builds the reference between the static model and the dynamically recorded protocol of the executed items. These identifiers are automatically generated.

The meta syntax of the following grammar is easy to understand: terminals are quoted, "empty" is the empty sequence of terminals, and each production ends with a dot.

```
Program          = Identifier StatementBlock.
StatementBlock   = Identifier "{" StatementList "}".
StatementList    = Statement StatementList | empty.
Statement        = Identifier
                   ( PrimitiveStatement
                   | TerminateStatement
                   | WhileStatement
```

```
                    | IfStatement
                    | SwitchStatement
                    | TryStatement )
                    SubExpressions.
IfStatement       = "if" "(" BoolExpression ")"
                    "then" StatementBlock
                    "else" StatementBlock.
WhileStatement    = "while" "(" BoolExpression ")"
                    StatementBlock.
SwitchStatement   = "switch" CaseHandler.
CaseHandler       = "case" StatementBlock CaseHandler
                    | "default" StatementBlock.
TryStatement      = "try" StatementBlock CatchHandler.
CatchHandler      = "catch" StatementBlock CatchHandler
                    | empty.
PrimitiveStatement = "stmt".
TerminateStatement = "throw" | "return" |
                     "break" | "continue".
```

The decision of an if statement and the loop condition of a while statement are Boolean values, and therefore, they are represented in the model to be used in logic-based coverage metrics such as MC/DC or term coverage. In contrast, the numerical expression of the switch statement and the exception types in the catch part of the try statements are not represented because no coverage metric is based on these types. The following second part of the language covers the Boolean and conditional expressions:

```
Expression      = BoolExpression
                  | ConditionalExpression.
BoolExpression  = Identifier
                  ( Condition | CompoundExpression ).
Condition       = "expr" SubExpressions.
CompoundExpression  = ("andThen"|"orElse"|"and"|"or")
                      "(" BoolExpression ","
                                BoolExpression ")".
```

```
ConditionalExpression = Identifier BoolExpression "?"
                        SubExpressions ":" SubExpressions ).
SubExpressions = "[" ExpressionList "]".
ExpressionList = Expression ";" ExpressionList | empty.
```

Because *BoolExpression* and *ConditionalExpression* are referenced in metric definitions, they have an identifier like the *Statement* and *StatementBlock*. The definition of *Condition* complies with [9]: "A Boolean expression containing no Boolean operators". The compound Boolean expressions like "A orElse B" are handled by *CompoundExpression*. For this, the model includes the (binary) tree structure of the Boolean expression's derivation tree. Thanks to the distinction between the operands *and* and *andThen*, the so-called short-circuit behavior can be applied to the GBT metrics. The *SubExpressions* are used to handle GBT-relevant expressions that are embedded in a primitive expression or statement. For example, let us consider the following Java expression:

```
A && f(B && C)
```

According to the definition of Condition, the complete term f(…) is a (primitive) condition. In order not to "lose" the embedded expression B && C, it is handled as an element of the embedded SubExpressions of the condition f(…). For determining expression-based metrics like MC/DC, these embedded expressions are also taken into consideration.

In the following example, a factorial function is transferred from Java into RPR. The unique identifiers are built with "S" for statements, "B" for statement blocks and "E" for expressions, followed by an ascending number.

Java	RPR
`int factorial(int n) {`	`P1 B1 {`

`  if(n < 0 || n > MAX) {`	`  S1 if(`
	`   E1 orElse(E2 expr [],`
	`    E3 expr [])`
	`     ) then B2 {`

`    return -1;` `  }`	`      S2 return []` `  } else B3 { } []`
`  int result = 1;`	`  S3 stmt []`
`  while (n > 1) {`	`  S4 while(E4 expr []) B4 {`
`    result *= n;`	`  S5 stmt []`

`    n--;`
`  }`
 	
`  return result;`
`}`
	`  S6 stmt []`
	` } []`
	
	` S7 return []`
	`}`

One major advantage of RPR is that the transformation of a given real language into the model can be clearly defined by mapping the right sides of the grammar productions. The mapping of a Java if statement, for example, into the model language can be described is as follows:

Java	RPR
"if" "(" Expression ")" Statement ["else" Statement]	"if" "(" BoolExpression ")" "then" StatementBlock "else" StatementBlock

It is easy to see that the Java "statements" in the *then* and *else* parts are mapped into statement blocks in the GBT model. In addition, the else part in Java is optional while the else part in the GBT model is not. If an if statement in the original program has no else block, it is added as an empty block in the GBT model.

2.2 Execution semantics

Every (primitive) GBT item such as a primitive statement or a primitive Boolean expression is described as a place-bordered and token-preserving Petri (sub)net called GBT model net (or short: model net). The model net of a primitive statement is shown in **Fig. 2**, the model net of the primitive Boolean expression in **Fig. 3** on the left side.

Loosely speaking, a model net is a Petri net with distinguished input and output places. Each model net has exactly one input place and one or more output places for normal and abrupt completion. The input place has no input transition relative to the model (sub)net. The output places have no output transition, respectively. In **Fig. 2**, **Fig. 3**, and **Fig. 4** these input and output places are located on the dashed border line of the GBT item. The initial marking of a model net has exactly one token in the input place while all other places are empty. Due to the token-preserving property the model net's end marking has exactly one token in one of the net's output places.

Because all model nets are place-bordered and token-preserving, they can be abstracted into both sub nets, which are reduced to the border places and super places [22]. **Fig. 2** shows this abstraction of a statement's model net. The abstractions are

used for theoretical net analysis like the reachability analysis as well as for describing the compound model nets of the complex GBT items.

In **Fig. 2** the conflict in the model net between the transitions $t_{ENormal}$ and $t_{EAbrupt}$ and their common input place s_E models the non-determinism of the statement's execution behavior, in **Fig. 3**, between the three transitions t_{EFalse}, t_{ETrue}, and $t_{EAbrupt}$ of a Boolean Expression): On the basis of the model net, it is not decidable whether a statement or expression completes abruptly or which Boolean result the real expression returns. But this execution behavior can be observed in the program under test by adding so-called probes to the program code. **Fig. 3** shows this connection between the instrumented program code of a GBT item and its corresponding model net: The execution area of the model net abstracts the statically undecidable behavior of the real program's expression or statement. The places s_{CIn}, s_{CT}, s_{CF} and s_{CA} abstract the execution counters of the program under test and build a specification for the source code's instrumentation.

Fig. 2. (Petri)-model net for a primitive statement

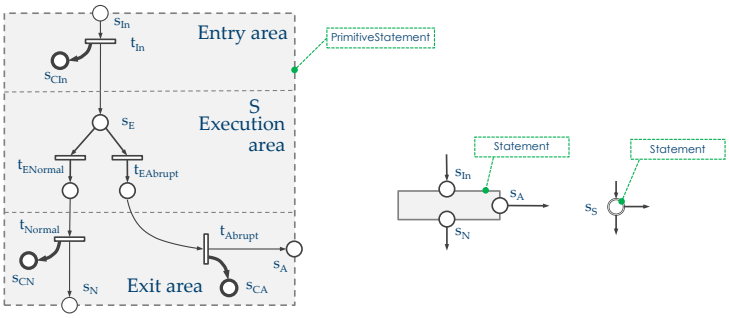
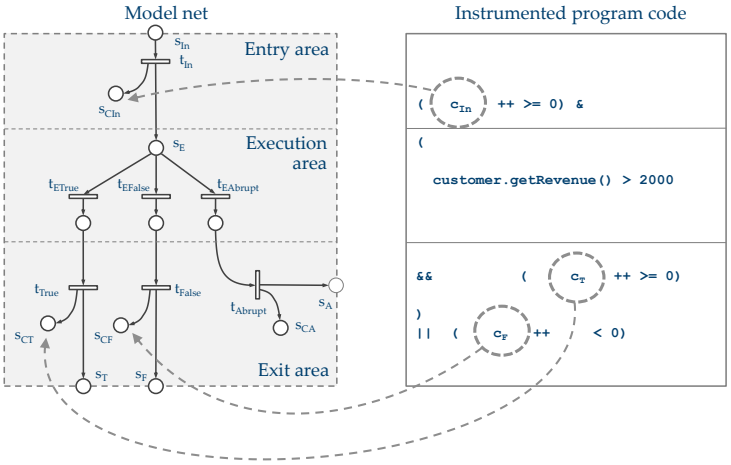


Fig. 3. (Petri)-model net for a primitive Boolean expression



The example of **Fig. 3** shows the instrumentation technique used by the GBT tool CodeCover [5, 6, 7]. In order to avoid additional method invocations, increment expressions using the so-called shortcut semantics "surround" the original expression. Details about this instrumentation can be found in [6].

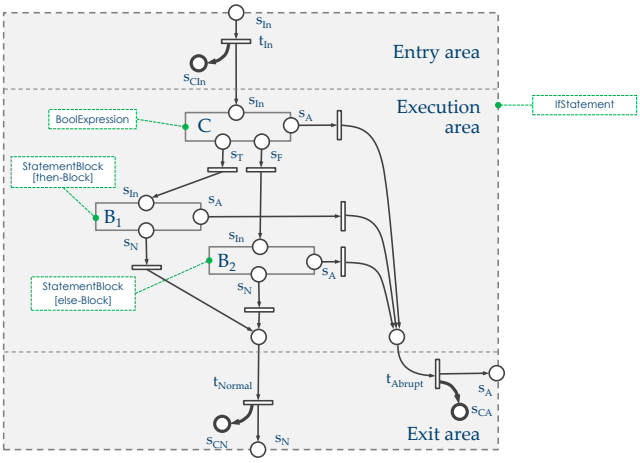
While the values of the "not-abrupt" execution counters c_{In}, c_T and c_F directly determine the markings of the model nets corresponding counter places, the value of the counter place for abrupt completion is not given by an instrumented counter. But due to the mathematically sound model net, after reaching a final marking, theese counter place markings can be derived:

$$| M(s_{CA}) | = | M(s_{CIn}) | - | M(s_{CT}) | - | M(s_{CF}) |$$

In the model net of the primitive expression in **Fig. 3** the proof is easy to see, for the more complex items the proof can be provided by using the Petri net's reachability analysis. Also by using the reachability analysis for all model nets of the various GBT items it can be proven that each (sub-)net is token-preserving and, as a consequence for each firing sequence starting from the initial marking a final marking is reached with exactly one token in one of the output places.

In contrast to the primitive GBT items the complex GBT items contain other GBT items as part of their own structure. E. g. an *if* statement contains a Boolean decision and a *then* and *else* statement block. **Fig. 4** shows the model net of such an *if* statement. Parts of this model net are embedded model (sub)nets of the embedded GBT-items which are abstracted into place-bordered boxes.

Fig. 4. (Petri)-model net for an if statement



This embedding technique automatically provides a dominance relationship between the GBT items. The embedded items are direct dominated:

A ddom B $\Leftrightarrow$ the model net of B is directly embedded in the model net of A

As already described, an important aspect of the model net is that there is a formal basis for the execution counters which have been generally used since the early beginnings of coverage testing [13]. The net model provides both a precise placement for the counters for normal completion and a (sound) basis for calculating counters for abrupt completion. As a result, the execution state of a GBT item *A* for a given test case *t* – "*A* is executed by test case t at least once" – can now be defined:

$$exe(A, t) \Leftrightarrow | M(s_{CIn}(A)) | > 0$$

This ability to report the execution of a GBT item for a particular test case (and not only for the entire test suite) is called "test-case selective GBT". While most GBT metrics are defined for the entire test suite, many useful GBT evaluations like test suite reduction, selective regression test and development of new test cases require the test-case selective GBT.

2.3 Metric definition

On the basis of the GBT items and their execution property we can now clearly and precisely define the GBT metrics. E. g. the statement coverage for an execution of a given program P can be defined as the proportion of executed statements; stmts(P) is defined as the set of all statements of P. The precise definition of "statement" is given by the grammar of the model language.

$$A \in exeStmts\ (P, T) \Leftrightarrow A \in stmts(P) \wedge \exists t \in T : exe(A, t)$$

$$stmtCov(P, T) = \frac{| exeStmts(P, T) |}{| stmts(P) |}$$

While the definition of statement and statement coverage is relatively clear, branch coverage does not have a precise common definition within the testing community. In general, two interpretations of "branch" exist in the literature [1, 3, 13]:

1. Every edge of the CFG forms a branch
2. Only those edges of the CFG are branches whose origin is a node with more than one outgoing edge.

Following the more intuitive definition no. 2, we define those so-called fork statements that have the described forking characteristics like if statements, while statements, or switch statements. For every fork statement *S*, we define a weight $w(S) \rightarrow \mathbb{N} \setminus \{ 1 \}$, which is the number of forking branches and an execution value $e(S, T) \rightarrow \mathbb{N}$ with $0 \leq e(S, T) \leq w(S)$, which is the number of executed branches for *S* with a test suite *T*. Thus, the branch coverage is defined as follows:

$$S \in forkStmts\ (P) \Leftrightarrow S \text{ is a statement with forking branches}$$

$$branchCov(P,T) = \frac{\sum_{i \in forkStmts(P)} e(i,T)}{\sum_{i \in forkStmts(P)} w(i)}$$

For some good reasons we also include the try statement in the forking statements, even though there is no distinct forking point. Otherwise, it would be possible to achieve full branch coverage without executing the catch blocks. In addition to the branch coverage we propose block coverage which defines the degree to which statement blocks are executed, similar to the statement coverage. A statement block is defined within RPR and e. g. is a *then* or *else* block of an *if* statement, a procedure body, or a catch block. Again, P is a program and *T* is a test suite.

$$B \in exeBlocks\ (P, T) \Leftrightarrow B \in blocks(P) \wedge \exists t \in T : exe(B, t)$$

$$blockCov(P, T) = \frac{|\ exeBlocks(P, T)\ |}{|\ blocks(P)\ |}$$

Generally, block and branch coverage are very similar because nearly all branches flow into a statement block and vice versa. A difference exists in the while statement where the branch that leaves the loop does not lead to a statement block. The procedure body constitutes a statement block but is no branch. Compared to the branch coverage, the block coverage has some practical advantages:

- The definition is easier and hence the understanding for the tester and the implementation in a GBT tool are easier.
- Unlike branches, statement blocks have a concrete representation in the program code. This makes the visualization of the coverage in the code easier and clearer.
- For block coverage, it is clear that catch blocks are taken into account, while most definitions of branch coverage do not address exception handling.

Some other coverage metrics which address control flow in expressions are described in [15].

3 Related Work

Most of the coverage metrics have their beginning in Myers' book [14] and in Huang's article [11]. While there is a lot of work using the CFG for coverage testing topics, there is only little attention from the testing community to the CFG's disadvantages described in Section 1. In [21] Binder describes a GBT model based on "code segments" which covers both (traditional) control flow and control flow in expressions. But he does not present this model in detail and does not develop a theory based on it. Ammann, Offutt, and Hong develop a theory for logic expressions in [1], but do not cover control flow or GBT-relevant expressions like the conditional expression. Zhu, Hall, and May shortly cover the topic of transformation (structured) program code into a CFG, but do not develop a GBT-language like the RPR. A de-

tailed coverage tool investigation is conducted in [9]. The authors develop a test suite to test whether the performance of a structural coverage analysis tool is consistent with the structural coverage objectives in DO-178B [22]. While the authors describe a complete set of test cases concerning control flow, control flow in expression, and logic expressions, they do not develop a model that "abstracts" the given test cases.

Lui et al. define in [23] coverage metrics based on Petri nets in the context of workflow applications. Comparable to this work the authors combine predefined sub-nets, but their scope is limited on the workflow aspects.

4 CodeCover

CodeCover is an open-source GBT tool that was initially developed 2008 in a student project at the University of Stuttgart. More information about that project is available in [5, 6]. Although stable releases have been available since 2008, CodeCover is still being extended and improved. Many enhancements have been implemented over the last years. CodeCover was developed to support both relatively small student projects and large industrial projects. For small projects, CodeCover provides an easy-to-use Eclipse integration. For larger projects, CodeCover can be used with the popular "Apache Ant" build tool or in batch mode. All information about installation and usage can be obtained from the CodeCover web page [7]. Currently, CodeCover supports the languages Java, C and COBOL. The GBT metrics provided are the control-flow based metrics introduced in Section 2, the loop coverage, conditional expression coverage, and term coverage [19], which is very similar to MC/DC.

4.1 User Interface

The easiest way to use CodeCover for Java programs is using the Eclipse integration. A typical picture of the CodeCover-Eclipse user interface is shown in **Fig. 5**.

To use CodeCover in an existing Java project only the following three steps must be performed:

1. Open "Properties" and select the CodeCover page. The desired GBT metrics can be selected here.
2. In "Project Explorer" select the packages or classes to be included in the GBT evaluation. Open popup menu and select "Use For Coverage Measurement".
3. If JUnit tests are used, run the test cases with "CodeCover Measurement for JUnit", otherwise open the "Run Configurations" and select the CodeCover dialog. Select "Run with CodeCover".

4.2 Instrumentation and data flow

CodeCover works with so-called source-code instrumentation, where the execution counters for the GBT items are added into the program's source code. **Fig. 6** shows the activities and artifacts which are visible for a tester using the CodeCover batch

interface. The model representation of the program code is stored in a so-called Test-SessionContainer (TSC).

Fig. 5. CodeCover Eclipse integration

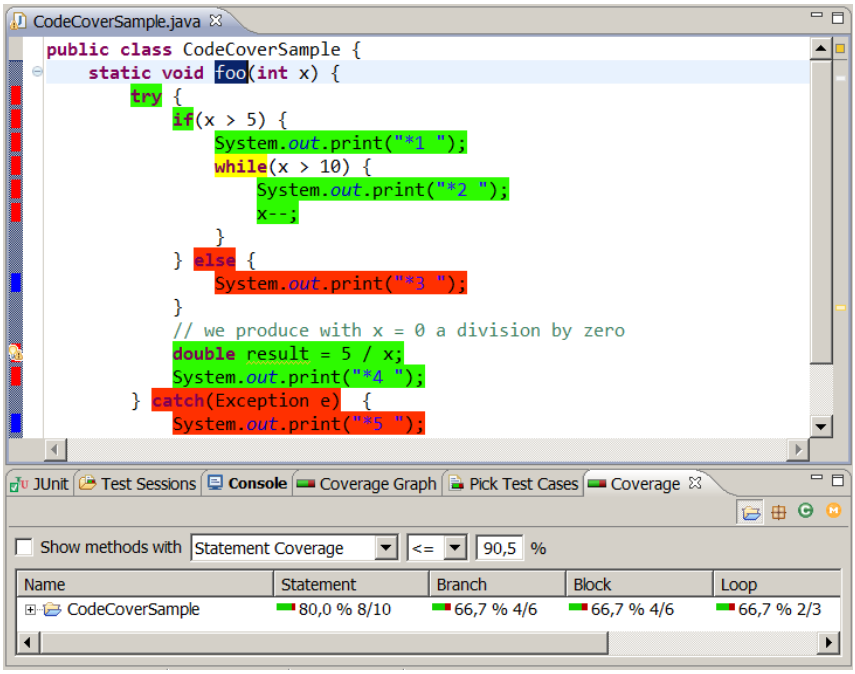
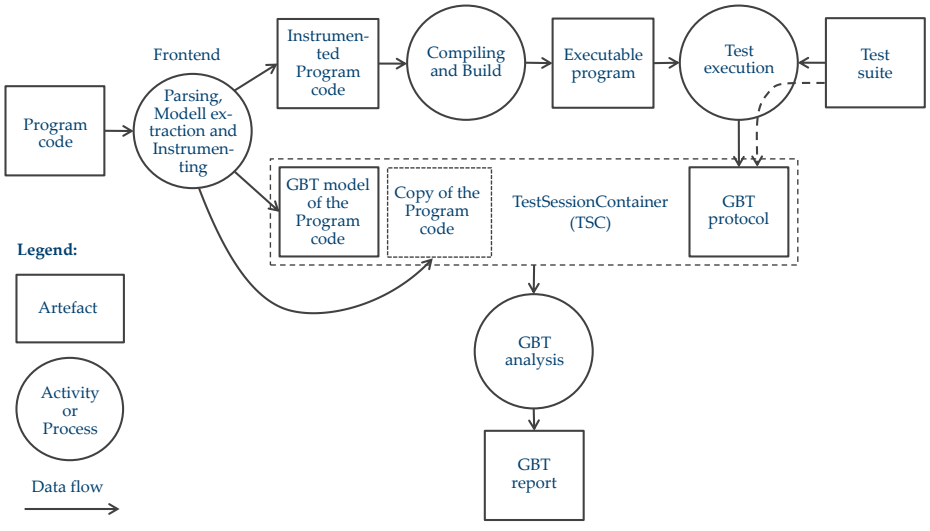


Fig. 6. Data flow

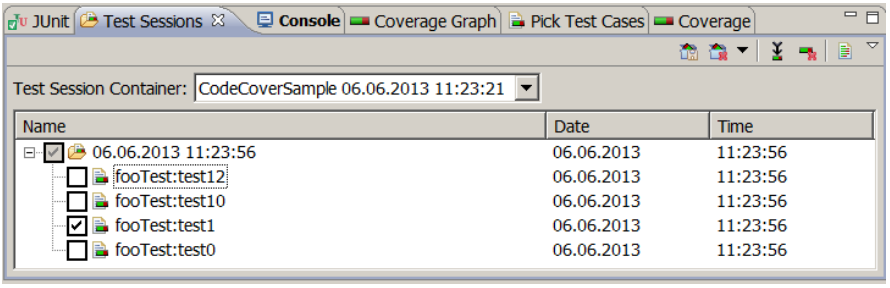


With the instrumentation for statement, block, branch, loop and term coverage, the compiled Java files are almost twice as large as before. For Java programs in practice, this “application blowup” usually is no problem. For C programs running on a micro controller, such growth is usually a problem and must be considered. After running the test cases of the entire test suite, the GBT protocol is added to the TSC. All GBT evaluations are based on the TSC.

4.3 Test case selective GBT

Beyond the functionality that most GBT tools offer, CodeCover can analyze the execution of individual test cases. In **Fig. 7** (“Test Sessions”) all the test cases of the entire test suite are listed. For a JUnit test, the identifiers of the test cases are derived from the JUnit test case names. Four test cases are executed but the user has selected only one test case (“fooTest:test1”). In this case the coverage visualization and summary report are generated only for this selected test case. Using this, the execution of a particular test case can be visualized. CodeCover can also list those test cases that executed an item the user selected by putting the cursor in the code area of the item. These test cases are listed in the view “Pick Test Cases” (**Fig. 8**). This function that is not found in other test tools provides useful information about the execution behaviour of the program under test.

Fig. 7. CodeCover view “Test Sessions”



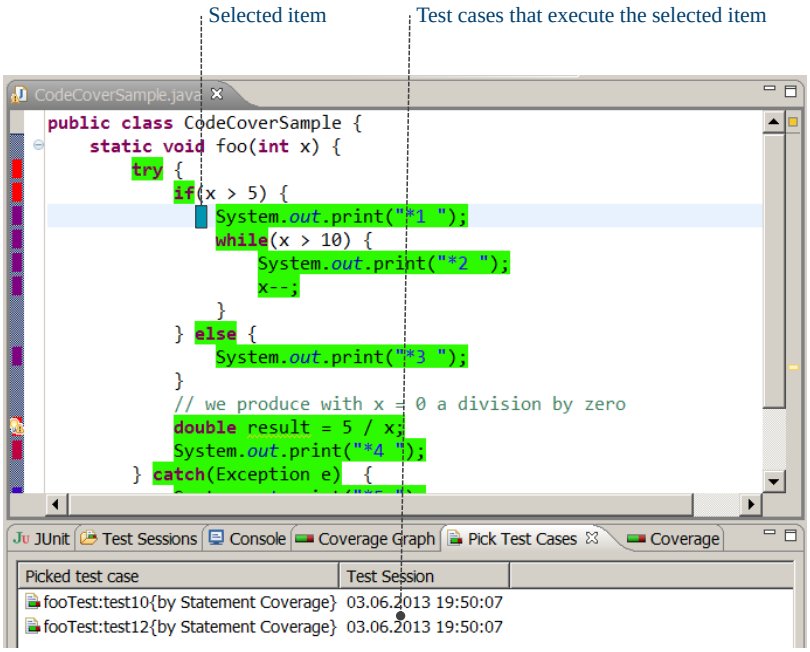
5 Conclusion

To date, most GBT metrics are defined either intuitively or based on the CFG. In the approach described above, the GBT metrics are precisely defined using a notation (RPR) which is applicable to a large class of programming languages. Expressions and exception handling are well integrated. Based on this model, the popular metrics are precisely defined. The tool CodeCover is an implementation that strictly follows these definitions.

When new GBT metrics are introduced, RPR can easily be extended to support the collection of these metrics. An example – and a topic for future work – are polymor-

phic expressions of object-oriented languages whose behavior is similar to that of switch statements.

Fig. 8. List of test cases that have executed a particular item



Acknowledgements

I would like to thank Prof. Jochen Ludewig for his valuable support and advice, as well as Kornelia Kuhle and the anonymous reviewers for their comments.

References

1. Ammann, P., Offutt, A.J., Hong, H.S., 2003, Coverage Criteria for Logical Expressions, Proc. International Symposium on Software Reliability Engineering, pp. 99-107, 2003.
2. Andrews, J. H., et al., 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Trans. Softw. Eng. 32, 8 (August 2006), 608-624.
3. Beizer, B., 1990, Software Testing Techniques, New York, Van Nostrand Reinhold
4. Berner, S., Weber, R., Keller, R. K., 2007, Enhancing Software Testing by Judicious Use of Code Coverage Information. In Proceedings of the 29th international conference on Software Engineering (ICSE '07).
5. Hanussek, R. et al., 2008, CodeCover - Glass Box Testing Tool, Design, Student Project "OST-WeST", University of Stuttgart, http://codecover.org/development/Design.pdf

6. Starzmann, M. et al., 2008, CodeCover - Glass Box Testing Tool, Specification, Student Project “OST-WeST”, http://codecover.org/development/Specification.pdf
7. CodeCover Homepage, http://codecover.org
8. Dupuy, A., Leveson, N., 2000, An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. Proc. Digital Aviation Systems Conference (DASC’00). Philadelphia.
9. Federal Aviation Administration, 2007, Software Verification Tools Assessment Study, DOT/FAA/AR-06/54, http://www.tc.faa.gov/its/worldpac/techrpt/ar0654.pdf
10. Fenton, N., Pfleeger, S. L., 1997, Software Metrics (2nd Ed.): A Rigorous and Practical Approach. PWS Pub. Co., Boston, MA, USA.
11. Huang, J.C., 1975, An Approach to Program Testing. ACM Comput. Surv. 7, 3 (Sep. 1975), 113-128.
12. IEC 61508. Functional safety of electrical/electronic/programable electronic (E/E/PE) safety related systems. Part 1–7, Edition 1.0
13. Mockus, M., Nagappan, N., Dinh-Trong, T.T., 2009, Test coverage and post-verification defects: A multiple case study. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09). IEEE Computer Society, Washington, DC, USA, 291-301.
14. Myers, G.J., 1979, Art of Software Testing, John Wiley & Sons, Inc., New York.
15. Schmidberger, R., 2013, Wohldefinierte Überdeckungsmetriken für den Glass-Box-Test. (Well-defined Coverage Metrics for the Glass Box Test; in German). Doctoral Dissertation; to be submitted to the Department of Informatics and Electrical Engineering, University of Stuttgart.
16. Yang, Q., Li, J.J., Weiss, D., 2009, A Survey of Coverage-Based Testing Tools. Comput. J. 52, 5 (August 2009), 589-597.
17. Zhu, H., Hall, P.A.V., May J.H.R., 1997, Software unit test coverage and adequacy. ACM Computing Surveys, 29(4):366–427, Dezember 1997.
18. Dahl, O.-J., Dijkstra, E. W., Hoare, C. A. R., 1972, Structured Programming, Academic Press
19. Chilenski, J. J., Miller, S. P., 1994, Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 9(5):193-200, September 1994.
20. Binder, R. V., 1999, Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
21. RTCA-DO-178B, 1992, Software considerations in airborne systems and equipment certification, December 1992.
22. Reisig, W.,1992, A Primer in Petri Net Design, Berlin, Springer-Verlag
23. Liu Z. et al., 2010, Test Coverage for Collaborative Workflow Application based on Petri Net, Proceedings of the 2010 14th International Conference on Computer Supported Cooperative Work in Design