Automatic Calculation of Coverage Profiles for Coverage-based Testing *

Raimund Kirner[1] and Walter Haas[1]

Vienna University of Technology, Institute of Computer Engineering,
Vienna, Austria, raimund@vmars.tuwien.ac.at

Abstract. Code-coverage-based testing is a widely-used testing strategy with the aim of providing a meaningful decision criterion for the adequacy of a test suite. Code-coverage-based testing is also used for the development of safety-critical applications, as the modified condition/decision coverage (MCDC) is proposed by the DO178b document. One critical issue of code-coverage testing is that they are typically applied to source code while the generated machine code may result in a different code structure due to code optimizations performed by an compiler. In this work we describe the automatic calculation of coverage profiles describing which structural code-coverage criteria are preserved by which code optimization. These coverage profiles allow to easily extend compilers with the feature of preserving any given code-coverage criteria by enabling only those code optimizations that preserve it.

1 Introduction

Testing is an established and accepted technique to increase the confidence in the correctness of a computer system. In contrast to formal verification, testing is not aimed to cover the full behavior of the system. But in contrast to formal verification, testing has the strong advantage that it operates on the real operation, including all low-level system details and physical behavior. Formal verification on the other side allways resides at a certain abstraction level, allowing the full behavioral coverage at this abstraction level. Thus, testing and formal verification are complementary approaches, both are necessary for the development of safety-critical systems.

Within this paper we focus on the testing part of verification, addressing the challenges towards portable test-data generation. Derivation or generation of test data is preferably done at the same level where the program is developed, typically a high-level programming language or any modeling environment with automatic code generation. First, this is the preferred way to do if the test data are written manually. Second, this is also beneficial for automatic test-data

* The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Sustaining Entire Code-Coverage on Code Optimization" (SECCO) under contract P20944-N13.

calculation, as it allows to reduce complexity by taking benefit of the abstract program representation. Thirst, this is preferred for portability issues, like cross-platform testing.

We assume that test-data generation is guided by structural code coverage, for example, *statement coverage*, *condition coverage*, or *decision coverage*. Structural code-coverage on its own is not a very robust coverage metric for software testing, but it is a useful complementary metric that indicates program locations of weak coverage by test data.

Using source-code based derivation of test data, it is the challenge to ensure that the test data fulfill an analogous structural code-coverage metric at the machine-code level as they achieve at source-code level. We call it analogous code-coverage metric, because several structural code-coverage metrics make no sense at machine-code level, because, for example, the grouping of several conditions to a decision is a source-level concept that is not available at machine-code level. If a compiler performs complex code optimizations that, for example, introduce new paths or change the reachability of some statements [1], this may disrupt the structural code coverage achieved at the original program.

We propose an approach toward the preservation of structural code coverage when transforming the program [2, 3]. For this we use a so-called *coverage profiles*, i.e., a pre-calculated table that specifies for each structural code-coverage metric which code transformations of the compiler guarantee to preserve it. Such a coverage profile can be easily integrated into a compiler such that only those code transformations are enabled that preserve the chosen structural code coverage. The conceptual integration of coverage profiles into a compiler is shown in Figure 1. In this paper we focus on the abstract specification of code transformations and on the calculation of the coverage profiles.

Besides the functional software testing, the preservation of structural code coverage is also of high interest for hybrid timing analysis, i.e., an approach to determine the timing behavior of a program based on the combination of execution-time measurements and program analysis [4, 5].

2 Structural Code Coverage for Software Testing

Structural code-coverage criteria are testing metrics to quantify the control-flow coverage of the program for a given set of test data. In this section we describe a few exemplary structural code-coverage metrics to show the calculation of compilation profiles. Formal definitions of some additional structural code-coverage metrics can be found in [3, 6].

2.1 Basic Definitions

In the following we give a list of basic definitions that are used to formally describe properties of structural code coverage and conditions for preserving structural code coverage:

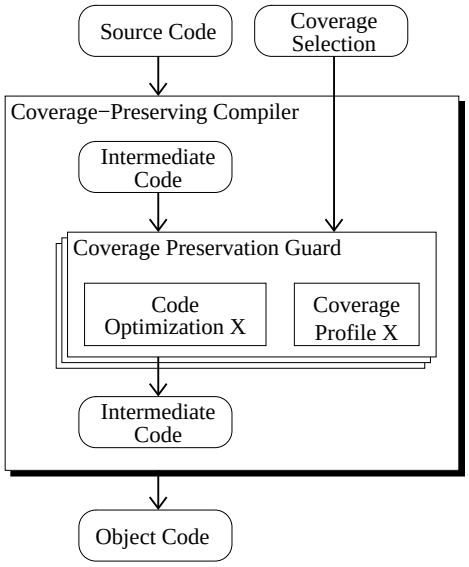

Fig. 1. Application of a Coverage Profile

Program P denotes the program before (P_1) and after (P_2) the transformations for which we want to preserve structural code coverage.

Control-flow graph (CFG) is used to model the control flow of a program [7]. A CFG $G = \langle N, E, s, t\rangle$ consists of a set of nodes N representing *basic blocks* (see below), a set of edges $E : N \times N$ representing the control flow (also called control-flow edges), a unique entry node s, and a unique end node t.

Basic block of a program P is a code sequence of maximal length with a single entry point at the beginning and with the only allowed occurrence of a control-flow statement at its end. We denote the set of basic blocks in a program P_i as $\mathrm{B}(P_i)$.

Decision is a Boolean expression composed of *conditions* that are combined by Boolean operators. If a *condition* occurs more than once in the *decision*, each occurrence is a distinct condition [8]. However, the *input* of a decision is the set of its conditions without duplicates. A decision is composed of one or more basic blocks. We denote the set of decisions of a program P_i as $\mathrm{D}(P_i)$.

There are programming languages, where decisions are hidden by an implicit control flow. For example, in ISO *C* due to the short-circuit evaluation the following statement `a = (b && c);` contains the decision `(b && c)`. The short-circuit evaluation of ISO *C* states that the second argument of the operators `&&` and `||` is not evaluated if the result of the operator is already determined by the first argument. The correct identification of hidden control flow is important, for example, to analyze decision coverage. See [3] for further details with respect to code coverage.

Condition is a Boolean expression. We consider only lowest-level conditions, i.e., conditions that do not contain operators with Boolean arguments [8]. A condition is composed of one or more basic blocks. We denote the set of conditions of a decision d as $\mathrm{C}(d)$. The set of all conditions within a program P_i is denoted as $\mathrm{C}(P_i)$.

Input data $\mathbb{ID}$ defines the set of all possible valuations[1] of the input variables of a program.

Test data $\mathbb{TD}$ defines the set of valuations of the input variables that have been generated with structural code coverage analysis done at source-code level. Since exhaustive testing is intractable in practice, $\mathbb{TD}$ is assumed as a true subset of the program's input data space $\mathbb{ID}$: $\mathbb{TD} \subset \mathbb{ID}$. If we would aim for exhaustive testing ($\mathbb{TD} = \mathbb{ID}$) there would be no challenge of structural code-coverage preservation.

Note that a test case consists, besides the test data, also of the expected output behavior of the program. Since we are primarily concerned with the preservation of structural code coverage with consider only the test data.

Reachability valuation $IV_R(x)$ defines the set of valuations of the input variables that trigger the execution of expression x, where x can be a condition, decision, or a basic block.

Satisfiability valuation $IV_T(x)$, $IV_F(x)$ defines the sets of valuations of the input variables that trigger the execution of the condition/decision x with a certain result of x: $IV_T(x)$ is the input-data set, where x evaluates to TRUE and $IV_F(x)$ is the set, where x evaluates to FALSE. The following properties always hold for $IV_T(x)$, $IV_F(x)$:

$$\begin{aligned} IV_T(x) \cap IV_F(x) &\supseteq \emptyset \\ IV_T(x) \cup IV_F(x) &= IV_R(x) \end{aligned}$$

Consider the following example of C code to get an intuition about the meaning of the satisfiability valuations:

```
void f (int a,b) {
   if (a==3 && b==2)
      return 1;
   return 0;
}
```

For this code fragment we assume

$$IV_R(\texttt{a==3}) = \{\langle a, b\rangle \mid a, b \in \texttt{int}\}$$

[1] Valuation of a variable means the assignment of concrete values to it. The valuation of an expression means the assignment of concrete values to all variables within the expression.

From this assumption it follows that

$$IV_R(\texttt{b==2}) = \{\langle 3, b\rangle \mid b \in \texttt{int}\}$$

(and not the larger set $\{\langle a, b\rangle \mid a, b \in \texttt{int}\}$ due to the hidden control flow caused by the short-circuit evaluation of ISO C [3]). It follows that

$$IV_T(\texttt{b==2}) = \{\langle 3, 2\rangle\}$$

Only those input data that trigger the execution of condition `b==2` and evaluate it to TRUE are within $IV_T(\texttt{b==2})$. With $\langle 3, 2\rangle$ the conditions `a==3` and `b==2` are both executed and evaluated to TRUE. Further, it holds that

$$IV_F(\texttt{b==2}) = \{\langle 3, b\rangle \mid b \in \texttt{int} \wedge b \neq 2\}$$

The definitions of $IV_R(x)$, $IV_T(x)$, and $IV_F(x)$ depend on whether the programming language has hidden control flow, for example, the short-circuit evaluation of ISO C [9].

2.2 Statement Coverage (SC)

Statement coverage (SC) requires that every statement of a program P is executed at least once. Statement coverage alone is quite weak for functional testing [10] and should best be considered as a minimal requirement. Using above definitions, we can formally define SC as follows:

$$\forall b \in \mathrm{B}(P).\ (\mathbb{TD} \cap IV_R(b)) \neq \emptyset \tag{1}$$

Note that the boundary recognition of basic blocks $\mathrm{B}(P)$ can be tricky due to hidden control-flow. A statement in a high-level language like ISO C can consist of more than one basic block. For example, the ISO C statement `f=(a==3 && b==2);` consists of multiple basic blocks due to the short-circuit evaluation order of ISO C expressions.

Remark 1. Source-line coverage is sometimes used as an alternative to SC in lack of adequate testing tools. However, without the use of strict coding guidelines, source-line coverage is not a serious testing metrics, as it is possible to write whole programs of arbitrary size within one source line.

2.3 Condition Coverage (CC)

Condition coverage (CC) requires that each *condition* of the program has been tested at least once with each possible outcome. It is important to mention that CC does *not* imply DC. A formal definition of CC is given in Equation 2.

$$\forall c \in \mathrm{C}(P).\ (IV_T(c) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F(c) \cap \mathbb{TD}) \neq \emptyset \tag{2}$$

Remark 2. Above definition of CC requires in case of short-circuit operators that each condition is really executed. This is due the semantics of $IV_T()$,$IV_F()$. However, often other definitions are used that do not explicitly consider short-circuit operators (as, for example in [11]), thus having in case of short-circuit operators only a "virtual" coverage since they do not guarantee that the short-circuit condition is really executed for the evaluation to TRUE as well as for the evaluation to FALSE.

2.4 Decision Coverage (DC)

Decision coverage (DC) requires that each *decision* of a program P has been tested at least once with each possible outcome. Decision coverage is also known as *branch coverage* or *edge coverage*.

$$\forall d \in \mathrm{D}(P).\ (IV_T(d) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F(d) \cap \mathbb{TD}) \neq \emptyset \tag{3}$$

3 Preservation of Structural Code Coverage

The challenge of structural code-coverage preservation is to ensure for a given structural code coverage of a program P_1 that this code coverage is preserved while the program P_1 is transformed into another program P_2. This scenario is shown in Figure 2. Of course if a program will be transformed, also the sets of basic blocks B or the set of program decisions D may get changed. As shown in Figure 2, the interesting question is whether a concrete code transformation preserves the structural code coverage of interest.

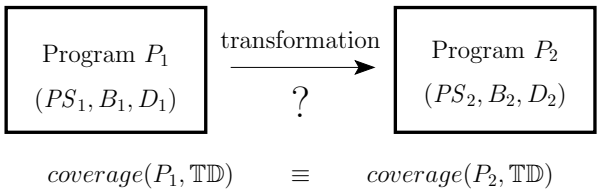

Fig. 2. Coverage-Preserving Program Transformation

When transforming a program, we are interested in the program properties that must be maintained by the code transformation such that a structural code coverage of the original program by the test-data set $\mathbb{TD}$ is preserved to the transformed program. Based on these properties one can adjust a source-to-source transformer or a compiler to use only those optimizations that preserve the intended structural code coverage. These coverage-preservation properties to be maintained have to ensure that whenever the code coverage is fulfilled at the original program by some test data $\mathbb{TD}$ then this coverage is also fulfilled at the transformed program with the same test data:

$$\forall \mathbb{TD}.\ coverage(P_1, \mathbb{TD}) \implies coverage(P_2, \mathbb{TD}) \tag{4}$$

In the following we present several coverage preservation criteria taken from [3]. We use these coverage preservation criteria together with abstract descriptions of the code transformations for the calculation of the coverage profiles.

3.1 Preserving Statement Coverage (SC)

Equation 5 of Theorem 31 provides a coverage preservation criterion for statement coverage. Equation 5 essentially says that for each basic block b' of the transformed program there exists a basic block b of the original program such that reaching b with a given test vector implies that also b' is reached with the same test vector.

Theorem 31 (Preservation of SC) *Assuming that a set of test data* $\mathbb{TD}$ *achieves* statement coverage *on a given program* P_1*, then Equation 5 provides a sufficient - and without further knowledge about the program and the test data (there is now knowledge about the test data or the program assumed), also necessary - criterion for guaranteeing preservation of statement coverage on a transformed program* P_2*. (Proof given in [3])*

$$\forall b' \in \mathrm{B}(P_2)\ \exists b \in \mathrm{B}(P_1).\ IV_R(b') \supseteq IV_R(b) \tag{5}$$

3.2 Preserving Condition Coverage (CC)

To define a coverage preservation criterion for CC (Theorem 32) we use the auxiliary predicate $touches\_ID(x, ID)$ given in Equation 6.

The predicate $touches\_ID(x, ID)$ is only True if the set of input data ID includes at least the true-satisfiability valuation $IV_T(x)$ or the false-satisfiability valuation $IV_F(x)$ of expression x, where x is either a condition or a decision. The predicate $touches\_ID(x, ID)$ is used for the coverage preservation criterion of CC (and also DC) to test whether the evaluation of any expression x of the original program to both, True and False, implies that the test data include at least one element of ID, needed for the coverage of an expression in the transformed program.

$$touches\_ID(x, ID) \quad \Rightarrow \quad (IV_T(x) \subseteq ID)\ \vee\ (IV_F(x) \subseteq ID); \tag{6}$$

Equation 7 states that for each condition c' of the transformed program there exists at least one condition of the original program whose coverage implies that c' evaluates to True and there exists at least one condition of the original program whose coverage implies that c' evaluates to False.

Theorem 32 (Preservation of CC) *Assuming that a set of test data* $\mathbb{TD}$ *achieves* condition coverage *on a given program* P_1*, then Equation 7 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of condition coverage on a transformed program* P_2*. (Proof given in [3])*

$$\begin{aligned} \forall c' \in \mathrm{C}(P_2).\ & \exists c \in \mathrm{C}(P_1).\ \mathit{touches\_ID}(c, IV_T(c')) \ \wedge \\ & \exists c \in \mathrm{C}(P_1).\ \mathit{touches\_ID}(c, IV_F(c')) \end{aligned} \tag{7}$$

3.3 Preserving Decision Coverage (DC)

To define a coverage preservation criterion for DC (Theorem 33) we use the auxiliary predicate $\mathit{touches\_ID}(x, ID)$ given in Equation 6, which is also used for preserving CC.

Equation 8 of Theorem 33 provides a coverage preservation criterion for decision coverage. Equation 8 essentially says that for each decision d' of the transformed program there exists at least one decision of the original program whose coverage implies that d' evaluates to TRUE and there exists at least one decision of the original program whose coverage implies that d' evaluates to FALSE.

Theorem 33 (Preservation of DC) *Assuming that a set of test data* $\mathbb{TD}$ *achieves* decision coverage *on a given program* P_1*, then Equation 8 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of decision coverage on a transformed program* P_2*. (Proof given in [3])*

$$\begin{aligned} \forall d' \in \mathrm{D}(P_2).\ & \exists d \in \mathrm{D}(P_1).\ \mathit{touches\_ID}(d, IV_T(d')) \ \wedge \\ & \exists d \in \mathrm{D}(P_1).\ \mathit{touches\_ID}(d, IV_F(d')) \end{aligned} \tag{8}$$

4 Automatic Calculation of Compilation Profiles

This section discusses the concepts and implementation behind automatic calculation of coverage profiles.

4.1 Program Model

For modeling control flow, the sequence of execution is defined by a set of labeled CFG edges $R : E \times \Lambda \times \Delta$, where $E : N \times N$ are the CFG edges with $N : B \cup C \cup \{s, t\}$, $\Lambda : \{T, F\} \times \{T, F, X\}$, and $\Delta : \{\delta_1, \ldots, \delta_{|R|}\}$. The special labels Λ and Δ are used to include information about control flow that depends on condition/decision evaluations and influence of input valuations.

Condition/decision labels $\ell \in \Lambda$ are used in case of condition nodes to determine the path a program uses when the control flow forks depending on the result of a condition evaluation. For flexibility in assigning condition results to different decision outcomes the condition/decision labels are two-parted. The first part defines the condition evaluation result using the symbols T and F for *true* and *false*. The second part of the label determines the decision result correlated with the condition result accumulated so far. It can be T, F or X if the decision outcome is not yet determined. Note that X is only allowed for edges originating and destinating inside the *decision hypernode*. All outgoing edges of a decision must carry a unique decision-label with T or F.

Each edge e_i in the graph is assigned a *valuation set* $\delta_i \in \Delta$. This valuation set represents all the valuations of the program input that trigger the execution of a path going through edge e_i. For each node v, except s and t, we have a continuity relation of the form

$$\bigcup_{e_i \in IN(v)} \delta_i = \bigcup_{e_j \in OUT(v)} \delta_j \tag{9}$$

where $IN(v)$ denotes the incoming edges of v and $OUT(v)$ the outgoing edges of v. In other words, execution paths entering a node must leave the node at least on one outgoing edge. The only exceptions are the entry-node s being the source and the exit-node t being the sink of each execution path.

4.2 Analyzing Code Optimizations

For analyzing the effect of code optimizations we model the valuation relations between the original and the transformed code. Based on the continuity relation (Equation 9) it is easy to obtain simple subset relations ($\subseteq$) between the valuation sets on incoming and outgoing edges inside each program graph. This can be done by walking through each node of the CFG and applying the continuity relation in forward and backward direction. These subset relations are the basic input for coverage preservation analysis.

A code transformation adds additional relations of valuation sets between the original and the transformed code, characterizing how the transformation forms the valuation sets of the edges in the transformed code based on the valuation sets of the edges in the original code. These relations can be propagated along the CFG based on the transitivity of subset relations.

4.3 The Mathematica Implementation

The implementation of the coverage-profile calculation was done using *Mathematica*, a fully integrated environment for technical and mathematical computing [12].

In a preparation-phase the control-flow graphs with the node sets B and C, the decision set D and the edge set R must be converted to the internal data structures of the program system. Each edge e in R is implemented as a tuple

$\langle v, w, \ell, \delta_i \rangle$ where v is the start- and w is the end-point of the edge. ℓ is the two-parted condition/decision label as described above (or empty if the edge is not originating at a condition node) and δ is a unique identifier for the valuation set.

Reachability and satisfyability valuation are reproduced internally by collecting the valuation sets on incoming and outgoing edges. $IV_R(x)$ is calculated as abstract union of all valuation sets on the incoming edges of node x. To calculate $IV_T(x)$ the union of all outgoing edges of x labelled with T are calculated and for $IV_F(x)$ all edges with label F are mentioned. Dependent wether x is a condition or a decision, the information is extracted from the condition or decision label.

We construct an auxiliary graph (derived from the CFG) for maintaining the equality relations ($=$) or subset relations ($\subseteq$) between valuation sets. The nodes of the auxiliary graph represent valuation sets or unions of valuation sets. A directed edge $\delta_i \rightarrow \delta_j$ is included in the support graph iff $\delta_i \supseteq \delta_j$ is true. In case of $\delta_i = \delta_j$ the auxiliary graph contains edges between δ_i and δ_j in both directions.

After constructing auxiliary graphs for the original code as well as the transformed code, these graphs are glued together by adding the additional relations caused by the code transformation. These subset relations form the abstract description of the code transformation that we use for the calculation of the coverage profiles. Creating a graph-reconstruction language [13, 14] that records the transformation relation while reconstructing the CFG is a possible extension for the future.

So far we have implemented preservation analyses for *statement coverage* (SC), *condition coverage* (CC), and *decision coverage* (DC). They get descriptions of the original CFG and the transformed CFG. Beside documentary information they output a verdict *true* or *false* about the ability of the transformation to preserve the mentioned coverage. The correctness of this verdicts relies on providing a correct and precise abstract description of the code transformation.

5 Examples of Analyzing Coverage-Preservation

This sections shows the coverage preservation analysis for several code optimizations. To avoid confusion when relating the valuation sets of the original code and the transformed code, we denote δ_i the valuation sets of the original code and ϱ_i the valuation sets of the transformed code. The results on coverage preservation are summarized in Table 1.

5.1 Condition Reordering with Short-Circuit Evaluation

Algebraic simplifications use algebraic properties of operators like associativity, commutativity and distributivity to simplify expressions [1]. Although these simplifications produce logically equivalent expressions, they may cause unexpected changes in the flow of control. Under certain circumstances these changes can

disrupt structural code coverage if they change the order of conditions. This is demonstrated in the following example of a branch with short-circuit evaluation.

The case study demonstrates condition reordering in an if-statement with two conditions connected by a logical AND operator with short-circuit evaluation. In a programming language the program code and the optimized code could look similar as in the following C-styled example:

```
if ( A && B )
   thenBlock
else
   elseBlock
```

changed to

```
if ( B && A )
   thenBlock
else
   elseBlock
```

Additionally, short-circuit evaluation of conditions is assumed, a technique used in several programming languages. In C/C++, e.g., logical expressions inside an if-statement are evaluated from left to right. If evaluation of further terms could not change the result anymore, evaluation stops and the branch is executed immediately. In the example, the second condition is not evaluated, if the result of the first condition evaluates to *false*.

Figure 3 shows the internal graph models for this use case with the original program on the left side and the transformed program on the right side. As a convention the symbols δ are used to note the valuation sets of the original program and symbols ϱ are used for the transformed program. In the original program, the described short-circuit branch is implemented with the edge from condition A to the else-block 4. In the transformed program the short-circuit branch connects condition B with else-block 14.

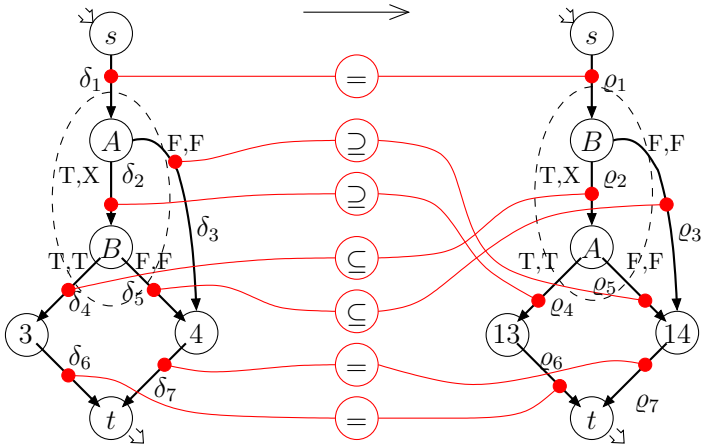

Fig. 3. Transformation Relation for Condition Reordering (with short-circuit evaluation)

Changing the condition order by swapping the conditions will not change the valuation-sets of the decision result. This is denoted by the equality relations (=)

between δ_6, ϱ_6 and δ_7, ϱ_7. Therefore, applying the preservation condition for DC (Equation 8) and for SC (Equation 5) will give a positive preservation verdict. But the distribution of the valuation sets of the conditions inside the decision are changed. Condition B in the transformed code snippet will now decide on a bigger valuation set than in the untransformed program while condition A in the transformed program decides on a subset of the possible valuations. Applying the preservation condition for CC (Equation 7) therefore results in a negative preservation verdict.

The sample output of the implemented analyzing function in Figure 4 shows how the function makes use of the preservation criteria to show, that statement coverage is preserved. The tool walks through each statement node of the transformed code. Using the continuity relation together with the additional subset relations on the valuation sets it determines those valuation sets which are a subset of valuation set $IV_R(x)$ of the currently investigated node x. Finally, it searches for a node in the original code with a valuation-set that is member of the related valuation-sets. In the first case this happens with node 3 and it's valuation set $IV_R(3) = \delta_4$. The same principle is used to find node 4 as a counterpart for node 14.

The last line of the listing gives as the function result the final decision, which is *true* in this case. This result can be used to be included into a coverage profile.

```
** SC-Preservation **
B(P2): {13, 14}
B(P1): {3, 4}
→→→ 1
IV_R( 13)== {ρ4} of P2 is related with
{{δ4} , {δ6} , {ρ4} , {ρ6}}
Nodes of P1 satisfying preservation condition: {3}
Accumulated scpf: True
→→→ 2
IV_R( 14)== {ρ3, ρ5} of P2 is related with
{{δ3} , {δ5} , {δ7} , {ρ3} , {ρ5} , {ρ7} , {δ3, δ5} , {ρ3, ρ5}}
Nodes of P1 satisfying preservation condition: {4}
Accumulated scpf: True
True
```

Fig. 4. Sample Output Analyzing Statement Coverage for a IF-Statement with Two Conditions (with short-circuit evaluation)

5.2 Loop Peeling

The transformation called *loop peeling* replaces the first k iterations from the beginning of a loop and inserts k copies of the body together with increment and test code of the loop index variable immediately ahead of the loop [1]. A

simplified example of this optimization is shown in Figure 5, where the compiler has peeled out the first iteration of the loop, placing one copy of the loop body and the loop termination test in front of the loop.

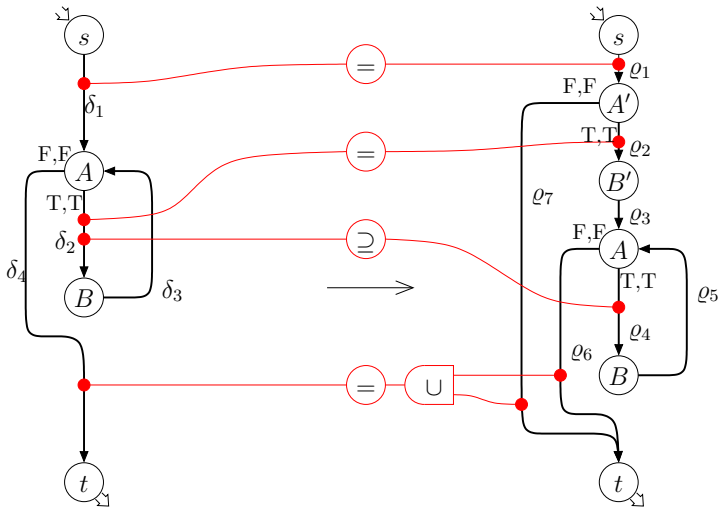

Fig. 5. Transformation Relation for Loop Peeling

From point of view of code coverage analysis, this little change in code structure has severe effects on preservation of all coverage criteria. In the original program SC, CC, and DC can be achieved by executing one iteration of the loop. After application of the transformation, the same test data will not enter the loop, because the first iteration has been executed in advance.

Formal analysis cannot prove coverage preservation, because the body of the loop is only triggered by a subset ϱ_4 of the original valuation subset δ_2. Therefore SC will fail for $b' =$ "B", because no statement b in the original program could be found such that $IV_R(b') \supseteq IV_R(b)$. Proofing preservation of CC and DC fails for similar reasons.

5.3 Loop Inversion

Loop Inversion, in source-language terms, transforms a `while` loop into a `do-while` loop [1]. The loop closing test is moved from the beginning of the loop to the end of the loop. In the simplest case this requires, that it is save to execute the loop body at least once. Otherwise, a test has to be generated in front of the loop to check the exit condition. This latter case is illustrated in Figure 6.

Although the relation of the valuation sets between the original and the transformed code contains many equalities, only statement coverage is preserved. This is, because the moved loop closing decision in the transformed program only decides on a subset of the input valuations compared with the original program,

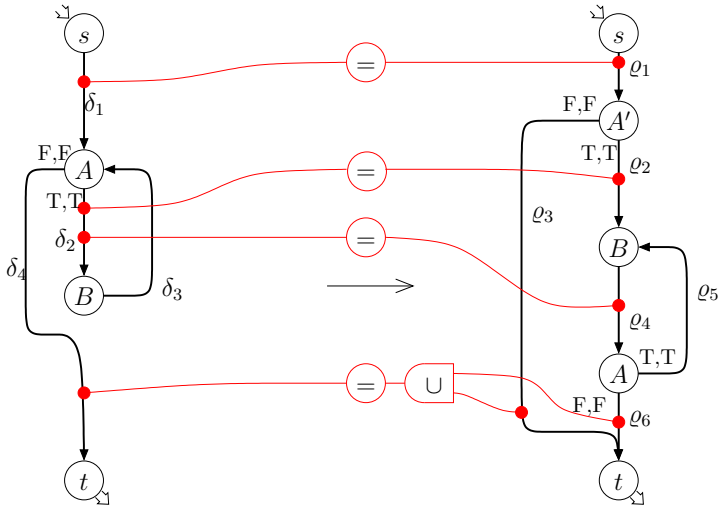

Fig. 6. Transformation Relation for Loop Inversion

which is expressed by the union operation ($\cup$) on the right side of the equality relation. This relation is induced by the subset-relation between ϱ_1 and ϱ_2.

5.4 Condition Reordering without Short-Circuit Evaluation

This example goes back to the condition reordering example presented in Section 5.1. The example presented in this subsection is a variation where all conditions are executed independently of the outcome of the other conditions of the decision. Besides SC and DC, also CC is now preserved. The main difference here is, that each condition decides on the full valuation-set $\delta_2 \cup \delta_3 = \varrho_2 \cup \varrho_3$, although the distribution between δ_2, δ_3 on one side and ϱ_2, ϱ_3 on the other side may differ.

The CFG in Figure 7 also shows an application for the two-parted condition/decision label. Although condition "B" in the original code on the left side can decide independently of the result of condition "A" for *true*, the decision result must be *false* if the result of evaluating condition "A" was *false*. The same is true in the transformed program when condition "A" decides *true* but the result of condition "B" was *false*.

Code Optimization	*Coverage Preservation*		
	SC	*CC*	*DC*
Cond. reordering (without short-circuit)	✓	·	✓
Cond. reordering (with short-circuit)	✓	✓	✓
Loop peeling	·	·	·
Loop inversion	✓	·	·

Table 1. Calculated Coverage Profiles

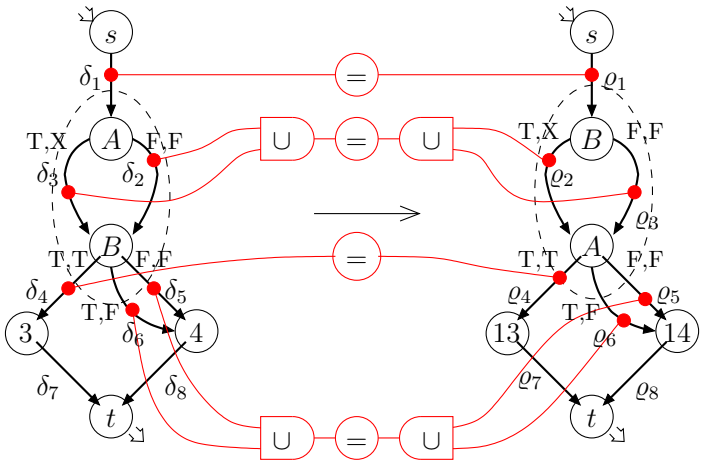

Fig. 7. Transformation Relation for Condition Reordering (without short-circuit evaluation)

6 Summary and Conclusion

In this paper we addressed the rather novel field of preserving structural code coverage during program transformation. A code transformer that take care of preserving structural code coverage has many interesting applications. For example, this allows the realization of reliable and portable test-data generators. Besides functional software testing, this is even interesting for measurement-based timing analysis.

Our approach is based on the calculation of so-called coverage profiles, which are tables that store the information of what code transformations guarantees the preservation of which structural code-coverage metric. To calculate these coverage profiles, we developed a formal coverage preservation criteria for each structural coverage metric and infer it with the abstract descriptions of the code transformations. We have calculated such coverage profiles for statement coverage (SC), condition coverage (CC), and decision coverage (DC).

As future work, we are focusing on extending the calculation of coverage profiles to more complex structural code-coverage metrics like the modified condition-decision criterion (MCDC) or a scoped path coverage.

References

1. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, Inc. (1997) ISBN 1-55860-320-4.
2. Kirner, R.: SCCP/x - a compilation profile to support testing and verification of optimized code. In: Proc. ACM Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07), Salzburg, Austria (2007) 38–42
3. Kirner, R.: Towards preserving model coverage and structural code coverage. EURASIP Journal on Embedded Systems **2009** (2009) doi:10.1155/2009/127945.

4. Wenzel, I., Kirner, R., Rieder, B., Puschner, P.: Measurement-based timing analysis. In: Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Porto Sani, Greece (2008)
5. Kirner, R., Puschner, P., Wenzel, I.: Measurement-based worst-case execution time analysis using automatic test-data generation. In: Proc. 4th International Workshop on Worst-Case Execution Time Analysis, Catania, Italy (2004) 67–70
6. Vilkomir, S.A., Bowen, J.P.: Formalization of software testing criteria using the z notation. In: Proc. 25th Annual International Computer Software and Applications Conference, Honolulu, Hawaii, USA (2001) 351
7. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1997) ISBN 0-201-10088-6.
8. Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Boeing Commercial Airplane Group (2001)
9. ISO: Programming Languages – *C*. 2nd edn. ISO/IEC 9899:1999 (1999) Technical Committe: JTC 1/SC 22/WG 14.
10. Myers, G.J.: The Art of Software Testing. John Wiley & Sons (1979)
11. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutoral on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, National Aeronautics and Space Administration, Hampton, Virginia (2001) available in pdf format.
12. Wolfram, S.: The Mathematica Book, 4th ed. Cambridge University Press (1999)
13. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. SIGPLAN Not. **37** (2002) 283–294
14. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM Press (2003) 220–231