Optimizing Compilation with Preservation of Structural Code Coverage Metrics to Support Software Testing

Raimund Kirner[1*], Walter Haas[2]

[1]*School of Computer Science, University of Hertfordshire, Hatfield, United Kingdom*
[2]*Institute of Computer Engineering, Technische Universität Wien, Vienna, Austria*

SUMMARY

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 mandated for the development of safety-critical applications, for example, the DO178b document requires the application of the modified condition/decision coverage (MCDC).
One critical issue of code-coverage testing is that structural code coverage criteria are typically applied to source code while the generated machine code may result in a different code structure due to code optimizations performed by a compiler. In this work we present the automatic calculation of coverage profiles describing which structural code-coverage criteria are preserved by which code optimization, independently of the concrete test suite. 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. Furthermore we describe the integration of these coverage profile into the compiler GCC. With these coverage profiles we answer the question of how much code optimization is possible without compromising the error-detection likelihood of a given test suite. Experimental results conclude that the performance cost to achieve preservation of structural code coverage in GCC is rather low.

Received ...

KEY WORDS: structural code coverage, source-level test-data generation, coverage preservation

1. INTRODUCTION

Testing is an indispensable process to reduce to reliability of software errors. Our focus here is to address the challenges associated with portable test-data generation. 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 [1].

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

*Correspondence to: University of Hertfordshire, Hatfield, AL10 9AB, UK. E-mail: r.kirner@herts.ac.uk

Contract/grant sponsor: "Sustaining Entire Code-Coverage on Code Optimization" (SECCO)/Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung); contract/grant number: P20944-N13. & "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering" (ADVANCE)/IST-FP7; contract/grant number: IST-2010-248828.

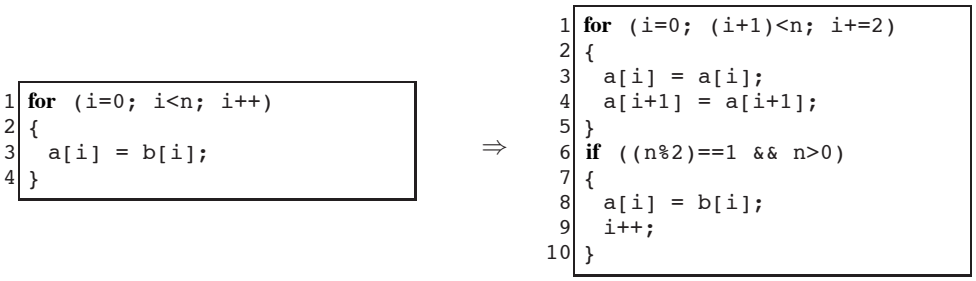


Figure 1. Code example for *loop unrolling* with unrolling factor $k = 2$: Creating two instances of loop body inside loop with extra code for the case that the iteration count of the original loop is not a multiple of k.

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 [2], this may disrupt the structural code coverage achieved at the original program.

To motivate the problem, lets look at *loop unrolling* as shown in Figure 1. In the given example, a test suite with {`n=2`} is sufficient to achieve statement coverage on the original code. However, after loop unrolling, the same test suite will fail to achieve statement coverage, as the remainder iterations at line 8 are not executed. Thus the transformed code contains statements that will not be tested with this test suite. Of course, the effect of the transformation on coverage preservation depends on the concrete test suite, e.g., a test suite with {`n=3`} will achieve full statement coverage of the transformed code. However, assuming zero-knowledge about the test suite, the example already shows that loop unrolling does not guarantee to preserve statement coverage.

We propose an approach toward the preservation of structural code coverage when transforming the program [3, 4]. For this we use a so-called *coverage profile*, i.e., a pre-calculated table that specifies for each structural code-coverage metric which code transformations of the compiler guarantee to preserve it for all possible input programs. This program-independence is an important benefit compared to the effort needed for testing-based correctness, e.g., as proposed for compiler debugging [5].

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 2. With this approach it is not necessary anymore to trouble about code instrumentation of the optimized code in order to measure the coverage. Instead one can measure the coverage for the non-optimized program, as our method then guarantees coverage preservation for the optimized program. With these coverage profiles we answer the question of how much code optimization is possible without compromising the error-detection likelihood of a given test suite.

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 [6, 7].

In our previous work we have formalized the structural code coverage metrics and developed necessary and sufficient conditions for the preservation of different structural code coverage metrics [4].

In this article we focus on the abstract specification of code transformations and on the calculation of the coverage profiles and its integration into a compiler. Section 2 discusses related work. In Section 3 we describe some basic concepts of program structure and structural code coverage criteria. In Section 4 we recapitulate the coverage preservation rules we introduced in [4]. The main contributions of this article start with Section 5. Here we describe the framework to calculate the coverage profiles. In Section 6 we demonstrate the calculation of coverage profiles with concrete examples. The integration of the coverage preservation into the GCC compiler and the performance evaluation is described in Section 7. Finally, Section 8 concludes this paper.

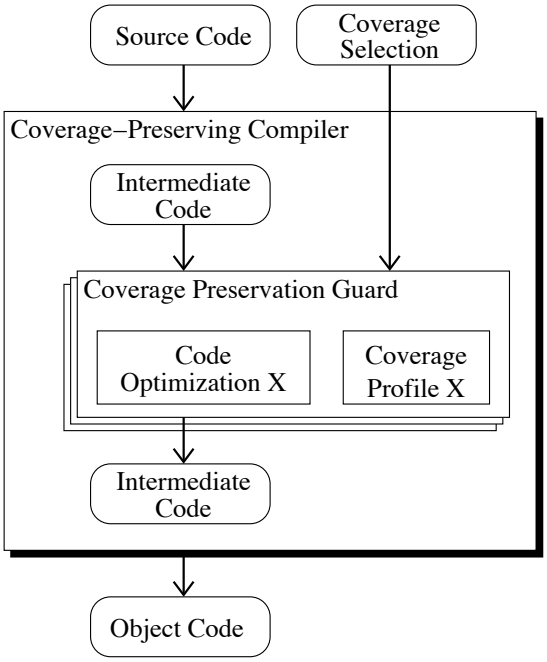


Figure 2. Application of a Coverage Profile

2. RELATED WORK

Structural coverage criteria are used as a supplementary criterion to monitor the progress of testing [38]. The DO178b document introduces the *modified condition-decision coverage* (MCDC) as a supplementary criterion for testing systems of safety-criticality level A [1]. Vilkomir proposes solutions to overcome some weaknesses of MCDC [39]. Vilkomir and Bowen have formally modeled structural code-coverage criteria using the Z notation [40]. The formalization we present in this article is basically equivalent, with the difference that we also support hidden-control flow [41], which is necessary to model code coverage for languages like ANSI C or ADA. Further, our notation is more compact, which has shown to be helpful for developing coverage-preservation criteria.

Model-based development aims to use high-level system representations within the system engineering process. For example, the Object Management Group proposes the *Model-Driven Architecture*, which explicitly differentiates between platform-independent and platform-specific models [42]. Models can be used to automatically generate source code. Another model-based approach is model-based testing where abstract models are used to guide the generation of test data [43, 44]. Using models to verify the correctness of the system requires evidence of the model's correctness [45].

Directly related to our work is the relationship of achieved model coverage and the resulting code coverage. Baresel et al. analyzed this relationship empirically, finding some correlation between the degree of model coverage and the resulting degree of code coverage [46]. Rajan et al. have shown for MCDC that the correlation of the degree of model coverage and the degree of code coverage depends on the code generation patterns [47]. With the approach presented in this article we are able to do better than relying on accidental coverage correlations, which makes our approach a significant improvement on testing safety-critical systems.

Elbaum et al. empirically studied the preservation of code coverage for software evolution with different change levels. They concluded that even relatively small modifications in the software may impact the coverage in a way that is hard to predict [48]. Their results also motivate our work for the preservation of code coverage.

A method complementary to our approach is described by Harman et al. They use testability transformation to produce a transformed program, which is then used by a test-data generator to improve its ability of generating test data for the original program [49].

A problem related to the preservation of code coverage during code transformation is the compiler correctness testing by Jaramillo et al. [5]. But while in the testing scenarios a high effort is required to ensure preservation of certain properties, our approach based on coverage profiles ensures preservation by design, i.e., it does not need any subsequent testing to show preservation of code coverage.

3. 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 structural code coverage preservation (SCCP) profiles.

3.1. 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:

Program P We want to study whether structural code coverage is preserved for a program P when applying some code transformations. P_1 is used to denote the program code before these transformations, and P_2 for the program code after these transformations.

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 as $\mathrm{B}(P)$.

Control-flow graph (CFG) is used to model the control flow of a program [8]. A CFG $G = \langle N, E, s, t\rangle$ consists of a set of nodes N representing *basic blocks*, i.e., $N = \mathrm{B}(P)$, 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. Note that the restriction to a unique end node has been chosen for simplicity reasons only; a formulation of coverage preservation with multiple end nodes would be straight-forward.

Program scope of a program P is a fragment of P with well-defined interfaces for entry and exit. We denote the set of program scopes in a program P as $PS(P)$. The general idea of program scopes is to define structural code coverage at a coarser level than single program instructions. Program scopes in its generic form are sets of program subpaths, where a subpath is any path within a program starting from an arbitrary program node leading to another arbitrary program node. The individual sets or program subpaths may even overlap. However, the concrete definition of a program scope is application specific. For example, we have used different definitions of program scope to trade the number of required test data against the number of required instrumentation points [9, 10].

Scoped path of a program scope ps is a sequence of control-flow edges from an entry point of the scope to an exit point of the scope. In case of nested program scopes, the whole inner program scope is a single block in the paths of the outer program scope. A scoped path of a program scope ps is uniquely represented by its starting basic block and the necessary TRUE/FALSE evaluation result of all conditions along the scoped path. We denote the set of scoped paths in a program scope ps as $\mathrm{PP}(ps)$. The paths within a program P, i.e., the scoped paths where the program scope subsumes the whole program, is denoted as $\mathrm{PP}(P)$.

Decision is a Boolean expression composed of logic predicates that are combined by Boolean operators. Regarding the control flow, a decision is composed of one or more basic blocks. We denote the set of decisions of a program P as $\mathrm{D}(P)$.

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. Further details with respect to code coverage can be found in an earlier description of the approach [4].

Condition is an logic predicate. The lowest-level predicates of a decision's Boolean expression are called *conditions*. A condition is composed of one or more basic blocks. If a *condition* occurs more than once in the *decision*, each occurrence is a distinct condition [11]. We denote the set of conditions of a decision d as $\mathrm{C}(d)$. The set of all conditions within a program P is denoted as $\mathrm{C}(P)$.

Input data $\mathbb{ID}$ defines the set of all possible valuations[†] 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 $IV_T(x)$, $IV_F(x)$:

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

Note that $IV_T(x) \cap IV_F(x)$ can be different to the empty set in case of a surrounding loop, allowing a single input-data item to cause both an evaluation to TRUE and to FALSE.

Consider the following example of ISO 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;
}
```

[†]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.

For this code fragment we assume

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

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* [4]). From this 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* [12].

3.2. 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 [13] 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.

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 typically possible to write whole programs of arbitrary size within one source line.

3.3. 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.

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

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, described by Hayhurst et al. [14]), 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.

For example, the following ISO C code fragment `if(a<=3||a>=5)` with the assumed input data $\{\langle a\rangle \mid a \in \{3, 5\}\}$ could be identified as incomplete CC according to Equation 2, but would be undefined given the description of CC used by Hayhurst et al. [14].

3.4. 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*.

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

3.5. Modified Condition/Decision Coverage (MCDC)

Modified Condition/Decision Coverage requires to show that each condition can independently affect the outcome of the decision [1]. Thus, having n conditions in a decision, $n+1$ test cases are required to achieve MCDC. Note that MCDC implies DC and CC. A formal definition of MCDC is given in Equation 4 based on the set of input test data $\mathbb{TD}$. It requires that for each condition c of a decision d there exists two test vectors such that the predicate symbol $unique\_Cause(c, d, td_1, td_2)$ holds, which ensures that the two test vectors show different outcomes for c as well as d but the same outcomes for all other conditions within d. This is exactly how MCDC is described above.

$$\forall d{\in}\mathrm{D}(P)\ \forall c{\in}\mathrm{C}(d)\ \exists td_1, td_2{\in}\mathbb{TD}.\quad unique\_Cause(c, d, td_1, td_2) \tag{4}$$

$$\begin{aligned} unique\_Cause(c_1, d, td_1, td_2) \quad \Rightarrow \quad & control\_Expr(td_1, td_2, c_1) \wedge \\ & control\_Expr(td_1, td_2, d) \wedge \\ & \forall c_2{\in}\mathrm{C}(d).\ (c_2 \neq c_1) \rightarrow \\ & \quad is\_invariantExpr(\{td_1, td_2\}, c_2) \end{aligned} \tag{5}$$

The predicate symbol $control\_Expr(td_1, td_2, x)$ tests whether one of the test data td_1, td_2 is a member of the input data set $IV_T(x)$ and the other one a member of the input data set $IV_F(x)$. If this predicate symbol is TRUE it is guaranteed that the expression x evaluates to both, TRUE and FALSE.

$$\begin{aligned} control\_Expr(td_1, td_2, x) \quad \Rightarrow \quad & (td_1{\in}IV_T(x)\ \wedge\ td_2{\in}IV_F(x)) \vee \\ & (td_2{\in}IV_T(x)\ \wedge\ td_1{\in}IV_F(x)) \end{aligned}$$

The predicate symbol $is\_invariantExpr(ID, x)$ tests whether the input-data set $ID \subseteq \mathbb{ID}$ provides a constant outcome for the evaluation of x. Actually, the predicate symbol $is\_invariantExpr(ID, x)$ is used to test whether there exists a test-data subset $\{td_1, td_2\}$ for a given condition, such that the results of all other conditions remain unchanged. Thus, this predicate symbol is used to ensure that each condition can independently control the output of the decision.

$$is\_invariantExpr(ID, x) \quad \Rightarrow \quad (ID \cap IV_T(x) = \emptyset)\ \vee\ (ID \cap IV_F(x) = \emptyset);$$

Above definition of MCDC is the original definition given in the RTCA/DO178b document [1], currently also called *Unique-Cause MCDC*. For the sake of code readability people often write their code in a deliberately redundant way such that 100% Unique-Cause MCDC cannot be achieved [11]. In order to be able to achieve 100% coverage, some weaker forms of MCDC have been specified: *Unique-Cause + Masking MCDC* and *Masking MCDC*. These alternative definitions of MCDC are described in [4]. For the sake of simplicity, in this article we only focus on the original definition of MCDC, i.e., Unique-Cause MCDC, as defined above. Extending above formal definition of MCDC to *Unique-Cause + Masking MCDC* or *Masking MCDC* is straight-forward. One has to exchange the predicate $unique\_Cause(c_1, d, td_1, td_2)$ by another predicate that formalizes the semantics of the alternative MCDC criterion.

As a comment on MCDC, even though MCDC also implies DC, there are cases where DC can find an error with a certain likelihood while MCDC can never find it. This is due to the specific way MCDC resembles DC [15]. Furthermore Kandl et al. have shown that the error detection performance of MCDC is quite different for three error categories [16].

3.6. Multiple Condition Coverage (MCC)

Multiple condition coverage (MCC) requires that each possible combination of outcomes of the conditions of each decision is executed at least once. MCC implies both DC and CC. MCC is not considered of high practical relevance, as it requires a rather large number of test cases: for n conditions in a decision 2^n tests are necessary.

3.7. Path Coverage (PC)

Path Coverage (PC) requires that each path of a program P has been tested at least once. Using our above definitions, we can formally define PC as follows:

$$\begin{aligned} \forall pp{\in}\mathrm{PP}(P) \quad \exists td{\in}\mathbb{TD}. \quad & (IV_R(\mathrm{B}_S(pp)) \cap \{td\}) \neq \emptyset \wedge \\ & \forall c_T{\in}\mathrm{C}_T(pp).\ (IV_T(c_T) \cap \{td\}) \neq \emptyset \wedge \\ & \forall c_F{\in}\mathrm{C}_F(pp).\ (IV_F(c_F) \cap \{td\}) \neq \emptyset \end{aligned} \tag{6}$$

Since the number of paths within a program typically grows exponentially with the program size (PC is even stronger than MCC), we do not consider PC to be a coverage metric of wide use in practice. However, due to reasons explained in Section 4.1, we will use the preservation criterion of PC as a safe approximation for the preservation criterion of scoped path coverage.

3.8. Scoped Path Coverage (SPC)

The scope of most of the structural code coverage metrics discussed so far basically spanned CFG edges between basic blocks (SC, CC, DC, MCDC, MCC), except PC, which is actually spanning across the whole program.

In order to define structural code coverage that goes beyond single CFG edges, but is well below the complexity of PC, we have defined the concept of program scopes (see Section 3.1).

Scoped Path Coverage (SPC) requires that each path within a program scope is tested at least once. Thus, there must be a test datum that covers all basic blocks along that path. Using our above definitions, we can formally define SPC as follows:

$$\begin{aligned} \forall ps{\in}\mathrm{PS}(P) \quad \forall pp{\in}\mathrm{PP}(ps) \quad \exists td{\in}\mathbb{TD}. \quad & (IV_R(\mathrm{B}_S(pp)) \cap \{td\}) \neq \emptyset \wedge \\ & \forall c_T{\in}\mathrm{C}_T(pp).\ (IV_T(c_T) \cap \{td\}) \neq \emptyset \wedge \\ & \forall c_F{\in}\mathrm{C}_F(pp).\ (IV_F(c_F) \cap \{td\}) \neq \emptyset \end{aligned} \tag{7}$$

Note, that the condition "$(IV_R(\mathrm{B}_S(pp)) \cap \{td\}) \neq \emptyset$" of Equation 7 ensures that in the pathological case of having a program scope that is completely free of conditions, coverage of the only single path in the program scope is guaranteed.

The number of required tests for SPC depends, besides the program complexity, also on the concrete definition program scopes, which can be defined in a free way to fit particular application needs.

4. 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 3. 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 3, the interesting question is whether a concrete code transformation preserves the structural code coverage of interest.

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 achieved by any test-data set $\mathbb{TD}$ is preserved at the transformed program. Based on these properties one can adjust

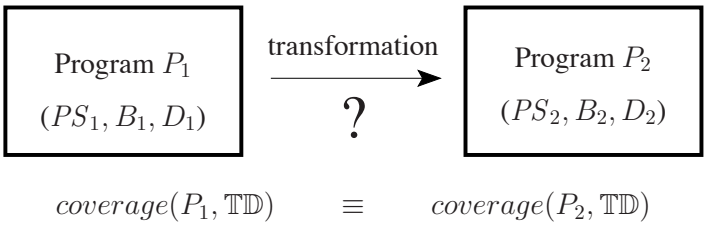


Figure 3. Coverage-Preserving Program Transformation

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 any test suite $\mathbb{TD}$ then this coverage is also fulfilled at the transformed program with the same test data:

$$\forall \mathbb{TD}.\ coverage(P_1, \mathbb{TD}) \implies coverage(P_2, \mathbb{TD}) \tag{8}$$

It is important to note that this characteristic of coverage preservation is independent of the concrete test suite $\mathbb{TD}$. In the following we present the coverage preservation criteria for PC and SPC. The coverage preservation criteria for SC, CC, DC, and MCDC are described in [4]. We use these coverage preservation criteria together with abstract descriptions of the code transformations for the calculation of the coverage profiles.

4.1. Preserving Path Coverage (PC)

To define a coverage preservation criterion for Path Coverage (Theorem 4.1) we use the auxiliary predicate $is\_CondTF\_enclosed(ID, C_T, C_F)$ given in Equation 9.

The predicate $is\_CondTF\_enclosed(ID, C_T, C_F)$ is only *True* if there is at least one condition from the set of conditions C_T whose true-satisfiability valuation is a subset of the input data ID or there is at least one condition from the set of conditions C_F whose false-satisfiability valuation is a subset of the input data ID. The predicate $is\_CondTF\_enclosed$ is used for the coverage preservation criterion of SPC to test whether for a condition in the transformed program with true/false-satisfiability valuation ID there exist two conditions in the original program whose true/false coverage are a subset of ID.

$$\begin{aligned} is\_CondTF\_enclosed(ID, C_T, C_F) \quad \Rightarrow \quad & \exists c_T \in C_T.\ IV_T(c_T) \subseteq ID \vee \\ & \exists c_F \in C_F.\ IV_F(c_F) \subseteq ID; \end{aligned} \tag{9}$$

As stated in Theorem 4.1, Equation 10 provides a coverage preservation criterion for PC. Equation 10 essentially says that for each path pp' of the transformed program there exists a path pp such that the reachability of the first basic block of pp implies the reachability of the first basic block of pp'. Further, Equation 10 states that for each condition c' of pp' that has to be evaluated to TRUE, there exists a condition c of a path in the original program that will imply the TRUE evaluation of c' (by predicate $is\_CondTF\_enclosed$). Finally, Equation 10 states that for each condition c' of pp' that has to be evaluated to FALSE, there exists a condition c of a path in the original program that will imply the FALSE evaluation of c' (by predicate $is\_CondTF\_enclosed$).

Theorem 4.1

(Preservation of PC) Assuming that a set of test data $\mathbb{TD}$ achieves *path coverage* on a given program P_1, then Equation 10 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of path coverage on a transformed program P_2. (Proof follows from the preservation proof of SPC by Kirner [4], as PS is the special case of SPC where there is only one scope that covers the whole program: "$pp \in \mathrm{PP}(ps) \wedge ps \in \mathrm{PS}(P)$" is simplified to "$pp \in \mathrm{PP}(P)$" in case of PC)

$$
\begin{aligned}
\forall pp' \in \mathrm{PP}(P_2). \quad & (\exists pp \in \mathrm{PP}(P_1).\ IV_R(\mathrm{B}_S(pp')) \supseteq IV_R(B_S(pp))) \wedge \\
& (\forall c' \in \mathrm{C}_T(pp')\ \exists pp \in \mathrm{PP}(P_1). \\
& \qquad is\_CondTF\_enclosed(IV_T(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp))) \wedge \\
& (\forall c' \in \mathrm{C}_F(pp')\ \exists pp \in \mathrm{PP}(P_1). \\
& \qquad is\_CondTF\_enclosed(IV_F(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp)))
\end{aligned} \tag{10}
$$

4.2. Preserving Scoped Path Coverage (SPC)

The preservation of scoped path coverage was one of our initial drivers for doing research on structural code coverage preservation. A formal definition of the coverage preservation criterion for SPC has been given by Kirner [4]. However, we finally ended up with the conclusion that without having the knowledge of the concrete definition of what a program scope is, it will not be possible to test coverage preservation in practice.

Thus, instead of applying a specific coverage preservation criterion for SPC we took the approach of applying the coverage preservation criterion for PC instead. Actually the coverage preservation criterion of PC is a sufficient but not necessary criterion for the preservation of SPC.

5. AUTOMATIC CALCULATION OF COMPILATION PROFILES

This section presents the automatic calculation of coverage profiles. The implementations of the coverage-profile calculations are based on search that directly follows the logic formulation of the coverage preservation criteria. Nearly all coverage preservation criteria presented in Section 4 or being described by Kirner [4] are based on set-theoretic relationships between input valuations. Therefore we designed a program model which provides a set of basic relationships between input valuations. This set is then used to construct the input-valuation relationships necessary to proof a certain kind of code-coverage preservation, using a one-by-one implementation of the formalism presented in Section 4. Thus, given the correctness proof of the formal coverage preservation criteria, little effort was required to come up with a correct implementation of the coverage-profile calculation.

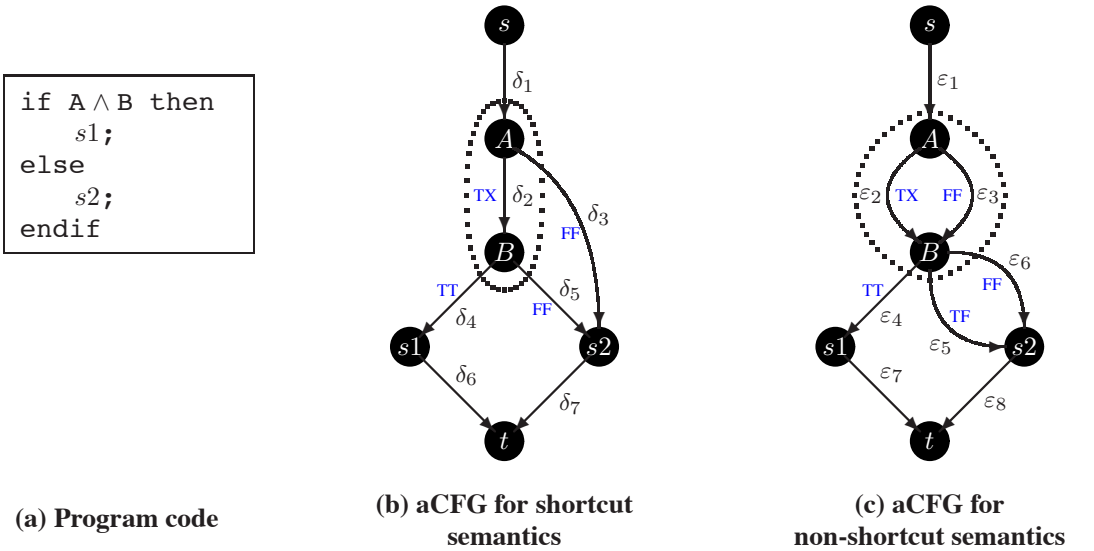
5.1. Program Model

Our code-coverage-analysis procedure internally uses an enhanced version of the CFG model [17] introduced in Section 3.1. CFGs are frequently used program representations in discrete event simulation and in different types of control-flow analysis and data-flow analysis (see, e.g., [18, 19, 20]). They are very illustrative objects and in most cases their structure can be derived directly from the program under investigation. Our enhancements of the CFG are specifically tailored for application of the preservation-analysis formalism introduced by Kirner [4] and in Section 4, and to support the implementation of the coverage-preservation criteria in the mathematical software system *Mathematica*. We will use the term *analysis control-flow graph* (abbreviated aCFG) to distinguish the enhanced CFG from classical CFGs.

Formally, an *analysis-control-flow graph* (aCFG) is a directed connected graph defined by the tuple $P := \langle B, C, D, R, s, t \rangle$, where $B \cup C \cup \{s, t\}$ is the node-set of the aCFG, D is a partition of C and R is a set of directed edges over $B \cup C \cup \{s, t\}$. Corresponding to the basic definitions in Section 3.1 the notation $B(P)$ refers to the set B of basic blocks, $C(P)$ refers to the set C of conditions and $D(P)$ refers to the set D of decisions.

Figure 4 illustrates the derivation of the aCFG model from the program code. It presents two possible implementations of a branch-statement. A pseudo-code representation of the investigated snippet of a program is shown in Figure 4(a). The branch-decision is composed of two conditions A and B linked by a Boolean *and*-operator. If the decision evaluates to TRUE, a statement-sequence $s1$ is executed. If the result is FALSE, execution continues with statement-sequence $s2$.

The first version of the aCFG shown in Figure 4(b) demonstrates an example of shortcut-evaluation semantics. Shortcut evaluation, also called short-circuit evaluation, is used in some programming languages to optimize the evaluation of Boolean expressions. The logical expression is only evaluated as far as necessary to determine a final result. In the present example the evaluation of condition B is skipped if evaluating condition A results to FALSE. In the aCFG there is a shortcut-edge $\langle A, s2\rangle$ to continue control-flow immediately with statement sequence $s2$ if execution of condition A results FALSE. In contrast Figure 4(c) implements the same piece of code but without shortcut-evaluation semantic. After evaluating condition A control flow always continues to evaluate condition B.



(a) Program code **(b) aCFG for shortcut semantics** **(c) aCFG for non-shortcut semantics**

Figure 4. Example of a branch statement and two possible representations by an aCFG, one with and one without shortcut-evaluation semantics. Input-valuation sets associated with the edges of the graph are denoted using the Greek letters δ, ε with an index.

The set B is called the *basic-block set* of the program P, because it comprises all nodes representing the non-conditional statements of the program or program fragment. The set C is called the *condition-set* of P. It comprises all nodes representing the conditions of the program. In the example Figure 4 the nodes $s1$ and $s2$ represent the operations inside the *then* and *else* branch. These are the only non-conditional statements, and therefore they are assigned to the statement set B. In both graph representations of Figure 4 the nodes labeled A and B represent the conditions, so the set C includes the nodes A and B.

The node s is the unique point of entry (*start node* or *entry node*), and t is the unique point of exit (called the *end node* or *exit node*) mentioned in Section 3.1. In principle, a program could have more than one exit node. But so far we did not found a need for that, and so we focus on aCFGs with a single exit node.

The set D, called the *decision set*, holds a partition of the condition set C into decisions and defines the assignment of the conditions to decisions. The sub-graph representing a decision $d_i \in D$ is treated as a *hyper-node* [21]. In Figure 4(b) and Figure 4(c), e.g., the condition nodes A and B together form the only one decision $\langle A, B\rangle$ of this example, which is the only one member of the set D. In the graph drawings we enclose such groups of conditions with a dotted line to emphasize the hyper-node representing the decision. We call edges with both endpoints being part of the same decision d_i *internal edges* of the decision. If one endpoint of an edge is member of d_i while the other endpoint is not, the edge is called an *external edge* of the decision. In Figure 4(b), for instance, the edge from condition A to condition B is an internal edge, while the edge from condition A to statement sequence $s2$ is an external edge of the decision $\langle A, B\rangle$. The order of conditions within a decision is defined by the internal edges of the decision.

The set R is the set of directed *control flow edges*. Mathematically every edge of an aCFG is defined as a tuple $e := \langle v, w, \alpha, \delta \rangle$. The pair v, w with $v, w \in B \cup C \cup \{s, t\}$ defines the endpoints and the direction of the edge. The node v is the origin of the edge and is called *head* of the edge, and w is the destination of the edge and is called *tail* of the edge. The head of an edge e is denoted $head(e)$ and the tail node of an edge e is denoted $tail(e)$ [22].

The component α is called the *condition/decision label* of edge e. If $head(e)$ is not a condition, the component α is empty. If $head(e)$ is a condition, the condition/decision label tags the outgoing edges of the node. Each condition includes a control-flow decision with a predefined set of possible results. Depending on the result of executing the condition control flow continues with the outgoing edge tagged with the result value. We use a two-parted condition/decision label, one label for the result of the condition and one label for the expected final result of the decision the condition is part of. Note, that our current implementation only deals with conditions and decision calculating the possible results TRUE or FALSE. We do not directly model statements with more than two branches like the `switch`-statement in C/C++, for example. Branches with more than two targets need to be modeled by a cascade of two-target branches.

In the aCFG drawing a two-letter combination denotes the condition/decision-label. The condition label is on the first position while the decision label is on the second position. The letter "T" means TRUE, and "F" means FALSE. Each outgoing edge of a condition node must be tagged with one possible results of the control-flow decision of the condition to find the correct successor where the flow of control continues dependent of the result of the condition execution. The decision label stores the possible result of the decision, as far as it is determined by the result of the currently evaluated condition. If the final result is not yet determined we use the symbol "X" as decision label. Otherwise, if the result of the decision is already determined, the decision label holds the final result of the decision. For example, if condition A in Figure 4(b) evaluates TRUE, the final outcome of the decision is determined by condition B, because of the *and* operation between the conditions. The other way round, when A evaluates to FALSE, condition B is not able to change this result. Therefore, the edge from condition A to condition B is tagged with "TX" to denote a condition outcome of TRUE with a yet still undefined outcome of the corresponding decision $\{A, B\}$. On the other hand, the edge from condition A to node $s2$ is tagged "FF" to denote, that the condition result FALSE also determines a final decision result FALSE. The symbol "X" is only allowed on internal edges of a decision. External edges of the decisions subgraph are required to be marked with concrete results of the decisions execution. E.g.: The external edge of decision $\{A, B\}$ from condition A to statement sequence $s2$ in Figure 4(b) is tagged "FF", because $s2$ is executed, if condition A evaluated to FALSE and the outcome of decision $\{A, B\}$ is FALSE in this case. However, as soon as the decision label on an outgoing edge of a condition represents a final result of the decision, no subsequent execution of the decisions conditions is allowed to change this result.

Our model also allows multiple edges between two nodes, as shown in Figure 4(c) for the connections from condition A to condition B and from condition B to statement $s2$. At first sight these double connections seem to be needles, because there is only one control-flow path. But the two paths are logically different, because the input valuations may differ dependent of the evaluation result of the condition. E.g., after evaluating condition A execution always continues evaluating condition B, independent of the result of executing condition A, but the input-valuation sets may be different, if condition A evaluates to TRUE or FALSE. Furthermore, continuation of the control flow after evaluating condition B does not only depend on the local result in node B. Only if condition B evaluates to FALSE then execution will continue with the *else* branch. But in case of result TRUE the control flow may continue in either directions dependent on the former result of condition A. Therefore two edges for condition result TRUE with different decision results are necessary to model the difference in the decision result.

The last component δ attaches a set of input valuations to the edge e. An input valuation $id \in \mathbb{ID}$ is member of δ, if it triggers an execution path that includes e at least once. Note, that δ is a symbolic value since no information is available about the members of the input-data set $\mathbb{ID}$ at analysis time. For clarification it should be emphasized, that our definition of an *execution path* allows to include an arbitrary number of instances of each edge. In graph theory this is often addressed as *walk* through

the graph [22]. In contrast to that, the common graph-theoretic definition of a *path* typically requires, passing each edge at most once.

Note, that the reachability valuation and the satisfyability valuation as defined in Section 3.1 can be transposed to the aCFG model in an obvious way by "summing up" the input-valuation sets on a certain kind of incoming or outgoing edges. The reachability valuation set of a node is the union of the input-valuation sets associated with the edges incoming to a node. The satisfyability valuation is determined in an analogous way. For conditions the satisfyability valuation is the union of all input-valuation sets associated with all outgoing edges marked with a certain condition label. When determining reachability valuation or satisfyability valuation of a hyper node representing a decision, the external incoming edges have to be used for calculating the reachability valuation, and the outgoing edges with a certain decision label have to be used for unifying the input-valuation sets to calculate the satisfyability-valuation set.

In Figure 4(b), for instance, the reachability valuation of $s2$ is $IV_R(s2) = \delta_5 \cup \delta_3$. The satisfyability valuations of condition A are $IV_T(A) = \delta_2$ and $IV_F(A) = \delta_3$ and the satisfyability valuations of the decision $\{A, B\}$ are $IV_T(\{A, B\}) = \delta_4$ and $IV_F(\{A, B\}) = \delta_3 \cup \delta_5$.

5.2. Analyzing Code Transformations

The principle of modeling relationships between input-valuation sets associated with the edges of the aCFG is inspired by a class of graph-theoretic problems known as *network flow theory* [23]. Instead of costs associated with the edges of the graph the valuations sets attached to each edge of an aCFG are used and set-theoretic operations replace the arithmetic calculations. To provide basic relationships between input valuations inside an aCFG we use a continuity property for input valuations. This property is comparable to *Kirchhoff's current law* in the theory of electrical networks. The property is based on the assumption, that each input valuation triggers a path sourcing at the entry node and terminating at the exit node. The only situation where some paths would not terminate at the exit node would be the existence of an infinite loop inside the investigated peace of program. In the current implementation we do not allow the existence of infinite loops. In addition, an execution path must form a feasible continuous sequence of statements according to the program's transition relation. In other words, the sequence of edges representing an execution path must not contain any gaps. The tail of an edge inside an execution path must be the head of its subsequent edge. The only exception is the edge ending in the termination-node.

Therefore, the input-valuation sets associated with the incoming-external edges and the outgoing-external edges of each node $v \in B \cup C$ must fulfill the following continuity property:

$$\bigcup_{i=1\ldots m_k} \delta_i = \bigcup_{i=1\ldots n_k} \varrho_i \tag{11}$$

where each δ_i denotes the input-valuation set associated to one incoming edge of the considered node v and ϱ_i denotes the input-valuation set associated with one outgoing edge of v.

Based on the continuity property for input-valuation sets described in Equation 11, it is easy to determine several non-strict superset relations between input-valuation sets associated with edges within the same aCFG. The principle is described below and illustrated in Figure 5. The determination procedure has to be performed for each node of the considered aCFG, except s and t. In the following description $\delta_1, \ldots, \delta_m$ denotes the input-valuation sets associated with the incoming external-edges of the inspected node, and $\varrho_1, \ldots, \varrho_n$ denotes the input-valuation sets associated with the outgoing external edges of the inspected node.

1. The equality
$$\delta_1 \cup \ldots \cup \delta_m = \varrho_1 \cup \ldots \cup \varrho_n$$
is given by the continuity property of Equation 11.
2. Using the equality (1), the following relations can be derived by set-theoretic considerations in forward direction:
$$\delta_1 \cup \ldots \cup \delta_m \supseteq \varrho_i \qquad 1 \leq i \leq n$$

3. Using equality (1), the following backward relations can be derived by set-theoretic considerations:

$$\varrho_1 \cup \ldots \cup \varrho_n \supseteq \delta_i \qquad 1 \leq i \leq m$$

Note, that the same procedure can be used for hyper-nodes representing decisions mentioning the external edges. In addition, a relation between the entry node and the exit node can be derived based on the assumption that all execution paths originate at the entry node and terminate in the exit node. So, there must be an equality between the union of the input-valuation sets associated to the incoming edges of the exit node and the input-valuation set associated to the outgoing edge of the entry node.

We call input-valuation relations inside one aCFG *local input-valuation relations* or *local relations* for short. Our procedure for determining such local input-valuation relations inspects all nodes except entry node and exit node, and the hyper-nodes representing the decisions of the inspected program. Walking through each node and hyper-node of the aCFG reveals a bundle of relations that can serve as one basis for code-coverage preservation analysis.

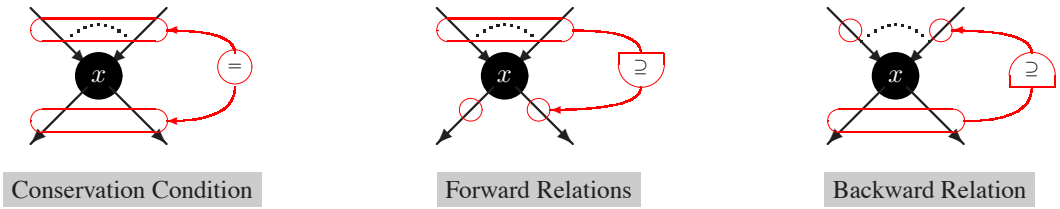


Figure 5. Determining input-valuation relations based on the continuity property.

The basic relations mentioned so far only consider input-valuation sets associated with edges incoming to and outgoing from the same node. Using the transitivity property of the superset-relation $\delta \supseteq \vartheta$ and $\vartheta \supseteq \varrho \Longrightarrow \delta \supseteq \varrho$ these basic relations can be expanded to get relations between input-valuation sets associated with edges not adjacent to one node. These input-valuation relations work equally for program structures inside a loop, as the input-valuations themselves do subsume all iterations of any surrounding loop.

Example:

Consider again the representation of the branch-statement with shortcut-evaluation from Figure 4(b) above. Inspecting the nodes $A, B, s1, s2$ and s, t reveals the following valuation-relations:

$$\begin{aligned}
\text{Node } A \quad &\Rightarrow \quad \delta_1 = \delta_2 \cup \delta_3 \\
&\qquad\quad \delta_1 \supseteq \delta_2 \\
&\qquad\quad \delta_1 \supseteq \delta_3 \\
\text{Node } B \quad &\Rightarrow \quad \delta_2 = \delta_4 \cup \delta_5 \\
&\qquad\quad \delta_2 \supseteq \delta_4 \\
&\qquad\quad \delta_2 \supseteq \delta_5 \\
\text{Node } s1 \quad &\Rightarrow \quad \delta_4 = \delta_6 \\
\text{Node } s2 \quad &\Rightarrow \quad \delta_3 \cup \delta_5 = \delta_7 \\
&\qquad\quad \delta_7 \supseteq \delta_3 \\
&\qquad\quad \delta_7 \supseteq \delta_5 \\
\text{Nodes } s, t \quad &\Rightarrow \quad \delta_1 = \delta_6 \cup \delta_7 \\
&\qquad\quad \delta_1 \supseteq \delta_6 \\
&\qquad\quad \delta_1 \supseteq \delta_7
\end{aligned}$$

Examples for additional relations obtained by transitivity of relations are $\delta_1 \supseteq \delta_4$ (using δ_2 or δ_6 as a link) or $\delta_1 \supseteq \delta_5$ (using δ_2 as a link).

□

An important prerequisite for analyzing the effect of code optimizations in a systematical manner is to keep track of the changes a transformation performed on the investigated piece of code. In our solution we do this by creating a second aCFG representing the investigated piece of code after the transformation has been performed, and by tagging the statements of the transformed program with supplementary information to keep a reference to the situation before the transformation. We call the aCFG, representing the transformed piece of code the *transformed* program and the aCFG representing the code before the transformation has been performed the *original* program.

The idea to create references pointing from the statements of the transformed program to statements of the original program is based on the observation, that a program transformation does not create the transformed program from scratch without considering the statements of the original program. To achieve the goal to tune the code for a given architecture, the code generator or compiler is free to transform the program in any way as long as the transformed program computes the same results as the original program specification. This can include reordering pieces of code as well as reducing the number of operations, replace them or insert new instructions [24]. This means that the code generator uses the statements of the original program as a kind of template to create the statements of the transformed program. In most cases statements of the transformed program are either an exact copy of the corresponding statements in the original program or they are modified versions of statements of the original program. Even newly created statements are not completely independent of the statements of the original program, since they are often added during the transformation to correct behavioral differences caused by modifications of some statements.

We say that a statement of the transformed program *performs the same function* as a statement of the original program, if the corresponding operations produce identical results for the same set of input data. But if the corresponding program-statements produce different results only for a subset of the input data then we say that the two operations *perform a similar function*. We call a pair of nodes representing statements performing the same function or a similar function *functional-related nodes*.

If two nodes, with one located in the original program and one located in the transformed program, perform the same function, then the input-valuations of the input-valuation sets associated to the incoming edges of the nodes are mapped to the input-valuation sets of the outgoing edges in the same way. Of course, this behavior is mandatory for non-conditional statements. Since nodes representing non-conditional statements have only one outgoing edge, changes of their functions cannot change the mapping of the input valuations from the incoming to the outgoing edges anyway. So the assessment of changes in control flow caused by program transformations is focused on nodes representing conditions and hyper-nodes representing decisions.

For nodes and hyper-nodes representing conditions and decisions, the mapping of the incoming input-valuations to the outgoing branches is defined by the calculated result of the control-flow decision. Therefore, the control-flow decision of the transformed program will be the same for the same set of input data, as long as the condition respective decision performs the same function as in the original program. On the contrary, if conditions or decisions are modified during a transformation, input valuations associated with the incoming edges of the corresponding nodes may be mapped differently to the outgoing edges, compared with the unmodified statement of the original program.

We characterize the notion of functional-related nodes and the two different levels of functional relationship more precisely in the following way. Let's assume a program transformation $P_1 \rightarrow P_2$. Also let v be a node or hyper-node of the original program P_1 which is functional related to the node or hyper-node v' of the transformed program P_2. We also assume, that both are representing a condition or a decision.

1. Two nodes v and v' are said to be *functional-equivalent nodes*, if their reachability-valuations and satisfyability-valuations meet the condition:

$$IV_R(v) = IV_R(v') \implies IV_T(v) = IV_T(v') \quad \text{and} \quad IV_F(v) = IV_F(v') \tag{12}$$

2. Two nodes or hyper-nodes v and v' are said to be *functional-similar nodes* if at least one of the satisfyability-valuation relations in Equation 12 is not an equality.

Please note, that Equation 12 also implies the weaker relations

$$\begin{aligned} &(1) \qquad IV_R(v) \subseteq IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \text{and} \quad IV_F(v) \subseteq IV_F(v') \\ &(2) \qquad IV_R(v) \supseteq IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \quad \text{and} \quad IV_F(v) \supseteq IV_F(v') \end{aligned}$$

In principle the same is true for functional-similar nodes, but the situation is more complex there. A statement about the relation between $IV_T(v)$, $IV_T(v')$ and $IV_F(v)$, $IV_F(v')$ respectively is not always possible when the equality between $IV_R(v)$ and $IV_R(v')$ changes to a non-strict superset or a non-strict subset. For example, consider the case, that for $IV_R(v) = IV_R(v')$ the satisfyability-valuations are related $IV_T(v) = IV_T(v')$ and $IV_F(v) \subseteq IV_F(v')$. If the prerequisite $IV_R(v) = IV_R(v')$ is dropped, only the following statements are valid:

$$\begin{aligned} &(1) \qquad IV_R(v) \subseteq IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \text{and} \quad IV_F(v) \subseteq IV_F(v') \\ &(2) \qquad IV_R(v) \supseteq IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \end{aligned}$$

So, if $IV_R(v) \supseteq IV_R(v')$ is true no relation can be calculated between the satisfyability valuations for $IV_F(v)$ and $IV_F(v')$, because $IV_F(v)$ is only a subset of $IV_F(v')$. If additional input valuations are added, then no general statement is possible whether $IV_F(v)$ is less, equal or greater than $IV_F(v')$.

Be aware that the concept of functional relations between nodes we used here to describe the effect of a program transformation does not necessarily imply a strong semantic identity of the statements represented by the pair of nodes. It just means, that both statements produce comparable results for the same input-valuations in the sense of distributing the input valuations in the same or in a similar manner. The program statements itself performing this task may be different. Consider for example a transformation as shown in Figure 6. It transforms a program by increasing the loops step-size from 1 to 2 and adapting the body accordingly. In both versions of the program the loop is entered for the same values of the variable N and possibly for an additional value, if N is an odd number. Therefore the loop decisions will still distribute the input-valuations in a similar manner. So the loop-statement of the original program and the transformed program will be classified as functional similar, although they are semantically different.

```
loop i ← 1 by 1 to N
    body(i);
endfor
```

```
loop i ← 1 by 2 to N
    body(i);
    if i+1 ≤ N then
        body(i + 1);
    endif
endfor
```

Figure 6. Pseudo code of a simple loop transformation producing two equivalent loops with semantically different but functional similar loop-statements.

With the help of the references in the transformed program the relationships between statements can be broken down to relations of the input-valuation sets associated to the edges of the transformed program and the input-valuation sets associated to the edges of the original program. In contrast to the local relations inside one aCFG we call these relations *inter-CFG relations*. Many of the inter-CFG relations can be derived systematically processing the nodes of the aCFG representing the transformed program in top-down order. The basic principle of performing this procedure for a single node or hyper-node is illustrated in Figure 7.

Since we assume that the remainder of the program is unchanged and therefore unaffected by the transformation, it follows that the input-valuation sets associated to the outgoing edges of both entry nodes are equal. This assumption is the basis for calculating other inter-CFG relations. With the help

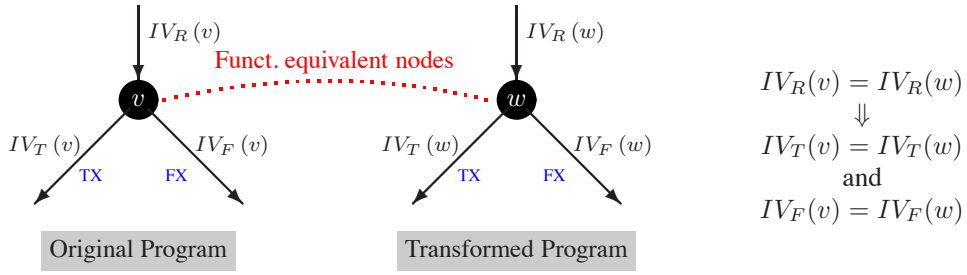


Figure 7. Determining inter-CFG relations by functional node relationships.

of local relations inside both aCFGs and already existing inter-CFG relations calculated in prior steps new relations can be created for the involved input-valuation sets of the investigated nodes of the transformed program and its related node in the original program.

The systematically determined relations are not always sufficient to be able to prove a certain kind of structural-code coverage. Especially relations that are determined by assumptions about the properties of a program transformation are not always systematically derivable. In these cases, additional relations derived from simple facts about the properties of the investigated transformation must be added manually to the set of relations. In most cases they are created using some prior knowledge about the behavior of a transformation.

Example:

To give an example for such a relation derived from the properties of a code transformation, consider a loop transformation like loop interchange, for instance. A basic requirement for loop interchange is that the same input must trigger the same number of iterations of the statements inside the innermost loop in the transformed program as well as in the original program. Therefore, in the original programs aCFG and the transformed programs aCFG the input-valuation sets associated to the edges entering the innermost loop must be equal.

5.3. Implementation of the Coverage Profile Calculation

The automatic analysis for code-coverage profile calculation was implemented with the tool *Mathematica* [25]. Since our analysis procedures are mostly one-by-one implementations of the formal coverage-preservation criteria, it was beneficial to use such a specialized mathematical software package which provides several built in functions for most of the extensively needed set-related data structures and operations. Our implementation of the code-coverage-preservation analysis comprises procedural implementations of the coverage-preservation conditions described by Kirner [4] and in Section 4. In addition we implemented some auxiliary functions to automatically reveal relations between input-valuation sets associated with the edges of the involved aCFGs as far as possible [26].

Prerequisite for the analysis is the existence of two aCFGs – one representing the structure of the investigated piece of code before the investigated transformation is performed and one aCFG to represent the structure of the same piece of code after performing the transformation. This pair of aCFGs is a general model for the investigated optimizing transformation. The structures of the aCFGs are derived from the way how the used compiler performs the investigated optimizing code transformation. It is not created from a particular source code or object code. Therefore, creation of the aCFGs and analysis of the code transformation has to be done only once for a particular compiler version. The compilation profile resulting from analysis of all code transformations can be reused as long as the compiler is not changed.

In addition to the pair of aCFGs some characteristic relations between the input-valuation sets of the original programs aCFG and the aCFG of the optimized piece of program are established together with the aCFGs. So the pair of aCFGs and the additional relations together represent the abstract description of the investigated code transformation.

Preparation of automatic analysis for a certain code transformation always starts with converting the given aCFG structures into *Mathematica* syntax [27]. In the current version this internal implementation of the aCFGs must be created manually. Figure 8, for instance, illustrates the internal representation of the aCFG from Figure 4(b).

To make sure, that the identifiers of the nodes are unique, we use integer numbers as internal representations for nodes. The external used names are attached to the nodes as labels, but we did not draw them in the graphs to keep the illustrations clearly arranged. Another internal detail of programming is that we do not explicitly specify the condition set C, as it is already specified by the members of the decision set D. Whenever needed, C is calculated from D.

```
B:={{"s1",3},{"s2",4}}
D:={{"decision1",{"A",1},{"B",2}}}
ST:={{"s",0},{"t",5}}
R:={{0,1,"",δ1},{1,2,"TX",δ2},{1,4,"FF",δ3},
     {2,3,"TT",δ4},{2,4,"FF",δ5},{3,5,"",δ6},{4,5,"",δ7}}
P:=eCFG[B,D,R,ST]
```

Figure 8. Internal representation of the aCFG structure given in Figure 4(b) above. The condition set C is omitted to avoid duplicate implementation of the same nodes in sets C and D.

When automatic calculation of coverage profiles is started, it first constructs an auxiliary graph for maintaining the equality relations ($=$) and subset relations ($\supseteq$) between input-valuation sets derived from the aCFG definition. This graph is similar to digraph models for relations [22]. The vertices of the input-valuation relation graph are the representations of the input-valuation sets involved. The arcs of the graph represent the subset relations implied by the basic properties of input-valuation relations. For example, if $\delta_i \supseteq \delta_j$ is true for some pair of input-valuation sets in the aCFG then a directed edge $\delta_i \rightarrow \delta_j$ is added to the auxiliary graph. In case of $\delta_i = \delta_j$ the auxiliary graph contains edges between δ_i and δ_j in both directions.

The structure of the *input-valuation-relation graph* represents the direct input-valuation relations gathered from continuity property Equation 11 and other directly derived relations. The nodes of the auxiliary graph represent valuation sets or unions of valuation sets. Please note, that this graph is not intended to be a complete representation of the relations. Especially it is not a transitive digraph, which would require that the existence of an arc $\langle x, y\rangle$ and $\langle y, z\rangle$ implies the existence of arc $\langle x, z\rangle$ [22]. Of course, the subset relations of the graph are transitive. But the intention of the input-valuation relation graph is contrary to that. Its purpose is to serve as an auxiliary structure supporting the search for transitivity relations based on already found basic relations "on the fly" whenever needed. A transitivity relation is found by searching for walks between two vertices of interest.

The construction of the input-valuation-relation graph must be done only once, because the information contained in the graph will not depend on the investigated coverage-preservation criteria. The graph will then be passed to the particular analysis function as one of several arguments.

After constructing an auxiliary graph for the original code as well as the transformed code, these graphs are glued together by adding additional relations derived from the properties of the investigated code transformation. Some of these relations can be derived automatically, other must be added manually. However, these relations between the input valuations of the original program and the input valuations of the transformed program form the abstract description of the code transformation that we use for the calculation of the coverage profiles.

After finishing the construction of the auxiliary graph the actual coverage-preservation analysis can be started. All analysis functions take the auxiliary relation graph constructed in the preparation phase, the aCFG of the original program and the aCFG of the transformed program as argument. So far we have implemented coverage-preservation analysis functions for *statement coverage* (SC), *condition coverage* (CC), *decision coverage* (DC), *modified condition/decision coverage* (MCDC) and *path coverage* (PC). 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.

To illustrate the principle function of the analysis, we will show here a simple example for useless code elimination, proving that it preserves statement coverage. *Useless code*, also called *dead code*, is reachable code which performs no computations that can affect final results [28]. In Figure 9 the statement `a:=3` in the second line obviously has no effect, since `a` is redefined in the third line. So the second line of the original program is removed by the optimizing compiler. Useless code is often the result of optimization steps passed before, although a program may include useless code before any optimization is applied to it [2].

Figure 9(b) and (c) shows aCFG representations for the original and the transformed program. Figure 9(b) shows the linear sequence of statements P_1 in its original structure. The input-valuation sets associated with the edges are denoted $\delta_1, \ldots, \delta_4$. Figure 9(c) shows the optimized structure P_2. Node 13, which is assumed to be useless code, is omitted in the transformed program and the edge originating at node 12 is directly connected with node 14. The edges of P_2 are associated with input-valuation sets ϱ_i to distinguish them from the input-valuation sets of P_1.

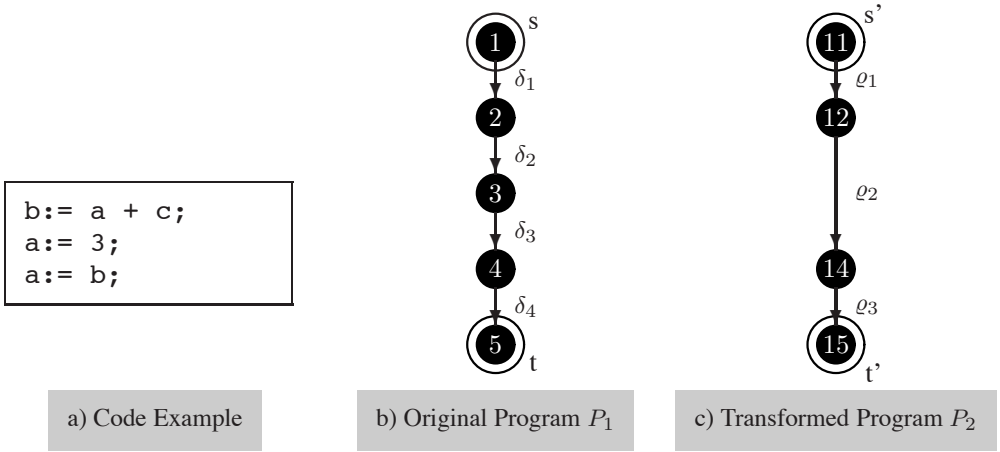


Figure 9. USELESS CODE ELIMINATION: Example code with useless assignment `a:=3` (a) and a pair of aCFGs representing the program before (b) and after (c) applying useless code elimination.

As further preparation steps for coverage-preservation analysis, the local relations of the relevant graphs have to be determined using the continuity property described in Section 5.2. In this case this is a very easy job, since there are only non-conditional statements involved. Walking through nodes 2, 3 and 4 of P_1 and the nodes 12 and 14 of P_2 reveals:

$$\underbrace{\delta_1 = \delta_2,\ \delta_2 = \delta_3,\ \delta_3 = \delta_4}_{P_1} \quad \text{and} \quad \underbrace{\varrho_1 = \varrho_2,\ \varrho_2 = \varrho_3}_{P_2} \quad \text{and} \quad \underbrace{\delta_1 = \varrho_1}_{P_1 \to P_2} \tag{13}$$

The transformation relation $\delta_1 = \varrho_1$ is true by default, since we assume that the remainder of the program is unchanged and therefore unaffected by the transformation.

Based on these relations, preservation of statement-coverage can now be proved using the preservation condition for statement coverage [4]. The condition requires that for all $b_2 \in B(P_2)$ a $b_1 \in B(P_1)$ exists with $IV_R(b_2) \supseteq IV_R(b_1)$. To prove the condition for some pair $\{b_2, b_1\}$, one has to find a path from $IV_R(b_2)$ to $IV_R(b_1)$ in the auxiliary relation graph representing the relation in Equation 13. If the proof succeed for all non-conditional nodes in P_2, statement coverage is proved for the optimizing code transformation.

In the example in Figure 9 the proof has to inspect nodes 12 and 14 in P_2. Let us assume, that we start with node 12. $IV_R(12) = \varrho_1$ can be calculated with the rules given at end of Section 5.1. We now search for some node in the set of the non-conditional nodes in P_1, which fulfills the preservation condition. Obviously, every non-conditional node of P_1 fulfills the condition. So, we take node 2 and calculate $IV_R(2) = \delta_2$. With the equality $\varrho_1 = \delta_1$ in Equation 13 the condition $IV_R(12) \supseteq IV_R(2)$ is proved. Continuing with node 14 of P_2 and taking again node 2 of P_1 we found

$IV_R(14) = \varrho_2 = \varrho_1 = \delta_1 = IV_R(2)$, and the preservation condition has been proved. An example for the possible output of the analysis functions is given at the end of Section 6.1 in Figure 11.

6. EXAMPLES OF COVERAGE PROFILE CALCULATIONS

This sections shows some examples of the coverage preservation analysis for several code optimizations [26]. Each of the use-case descriptions first provides a brief description of the applicability and the basic properties of the code transformation. For better understanding, a small code example is given to illustrate how the transformation works. For easier understanding we use a source-code like language for the code examples, although the optimizations may be performed on intermediate-code level or object-code level. Secondly, the transformation conditions and the basic facts for preservation analysis including aCFG models for the original and the transformed program are explained. In the third part, some facts about the properties of the investigated code transformation are described, as far as they are important for the coverage-preservation analysis. Finally, a summary of the analysis results is given with formal arguments, why a particular kind of coverage has been calculated as preserved or not preserved.

In the descriptions P_1 always refers to the original version of the program and P_2 refers to the transformed program. To avoid confusion when relating the valuation sets of P_1 and P_2 in the aCFG drawings, we use the notation δ_i for the valuation sets of the original code P_1 and ϱ_i for the valuation sets of the transformed code P_2. Additional circles denote the entry-node and the exit-node. In addition, the identifier with the lowest value of an aCFG is by convention associated with the entry-node, while the identifier with the highest value inside an aCFG is associated with the exit-node. Decisions are marked with a dotted oval enclosing all conditions the decision is composed of. A dotted line pointing from the node or hyper-node of the transformed program to some node or hyper-node of the original program emphasizes the functional relationship of a transformed statement in the original program to a statement of the original program. But these relationships are only drawn in this way, if they are important for the analysis. In addition, the string "$\mapsto X$" is sometimes placed beside a node of the transformed program, to point out that this node is identical with node X in the original program. This notation is used if the relation is important for better understanding of the use-case but is not of vital importance for the calculation of coverage profiles.

The results of all coverage-preservation use cases presented in this section are summarized in Table I.

6.1. *Condition Reordering with Short-Cut Evaluation*

Algebraic simplifications use algebraic properties of operators like associativity, commutativity and distributivity to simplify expressions [2]. 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-cut evaluation. Figure 10.a presents a source-code level example for reordering the conditions inside the logical expression of an if-statement. The compiler uses commutativity of the logical *and* operator to swap the conditions of the branch decision.

Condition reordering is an example of a code transformation that is not a classical optimization. It usually serves as a preparation step to apply further optimizing code transformations. Consider, for instance, an expression which contains two sub expressions $A \wedge B$ and $B \wedge A$. Reordering the operands in $B \wedge A$ to $A \wedge B$ reveals the existence of two identical sub expressions. This enables further application of optimizing transformations like algebraic simplifications or common sub-expression elimination.

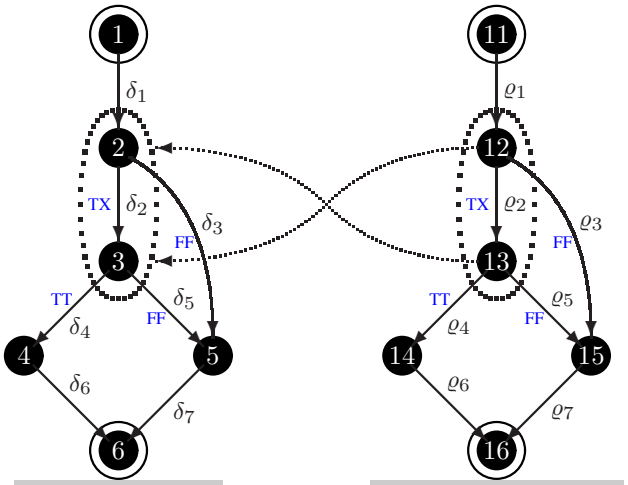
A possible aCFG structure for condition reordering is shown in Figure 10.b. The shortcut property of the condition evaluation is characterized by the edge $\langle 2, 5\rangle$ in the original program and $\langle 12, 15\rangle$ in the transformed program. Compared with the example from Figure 10.a, node 2 represents the condition "`a > 0`" and node 3 represents the condition "`a < 10`". If condition 2 evaluates to $False$, the second condition 3 is skipped, because the final result of the decision has been already obtained and

```
if (a > 0) ∧ (a < 10)
then
    then_statement;
else
    else_statement;
endif
```

⇒

```
if (a < 10) ∧ (a > 0)
then
    then_statement;
else
    else_statement;
endif
```

a) Possible code example for a reordering transformation of two conditions inside a branch decision.



b) aCFGs of swapping the conditions of a branch with two conditions evaluated with short-cut semantics and a non-empty else fork.

Figure 10. Transformation: CONDITION REORDERING

the result of the second condition is unable to change it. This behavior corresponds to the semantics of the operators && and || in ISO C/C++ [12, 29] and to the semantics of the operators *and then* and *or else* in ADA [30].

Since the connection $\langle 2, 5\rangle$ and $\langle 12, 15\rangle$ respectively can bypass the evaluation of the second condition, some execution paths can get excluded to the second condition in the transformed program. Consider, for example, a sequence of three test cases where the conditions evaluate to $\{TT, FF, TF\}$, with the first value of each pair assigned to condition 2 and the second assigned to condition 3. This set of test-data achieves condition coverage in the non-transformed program, covering all possible paths inside the decision. Due to the condition swap in the transformed program, the same set of test-data now achieves that the conditions evaluate to $\{TT, FF, FT\}$. Due to the short-cut semantic the path $11 - 12 - 13 - 15 - 16$ is not covered anymore, because the test cases where the conditions 12 and 13 would evaluate FF and FT both bypass node 13 over the shortcut edge. Condition coverage is therefore not achieved, because the second condition only goes into one direction.

The main property of the condition-reordering transformation is, that the statements in the *then* branch of the transformed program fragment must be entered for the same input values than in the original program. The same is true for the statements in the *else* branch. Therefore a valid condition reorder transformation must fulfill the following formal properties: $\delta_4 = \varrho_4$ for the *then* branch and $\delta_3 \cup \delta_5 = \varrho_3 \cup \varrho_5$ for the *else* branch.

** SC-Preservation **
B(P2): $\{13, 14\}$
B(P1): $\{3, 4\}$
$\rightarrow\rightarrow\rightarrow 1$
$\mathrm{IV}_R(13) == \{\rho_4\}$ of P2 is related with
$\{\{\delta_4\}, \{\delta_6\}, \{\rho_4\}, \{\rho_6\}\}$
Nodes of P1 satisfying preservation condition: $\{3\}$
Accumulated scpf: True
$\rightarrow\rightarrow\rightarrow 2$
$\mathrm{IV}_R(14) == \{\rho_3, \rho_5\}$ of P2 is related with
$\{\{\delta_3\}, \{\delta_5\}, \{\delta_7\}, \{\rho_3\}, \{\rho_5\}, \{\rho_7\}, \{\delta_3, \delta_5\}, \{\rho_3, \rho_5\}\}$
Nodes of P1 satisfying preservation condition: $\{4\}$
Accumulated scpf: True
True

Figure 11. Sample Output Analyzing Statement Coverage for a IF-Statement with Two Conditions (with short-circuit evaluation)

A sample output of the implemented automatic analysis function for Statement Coverage (SC) is given in Figure 11. The output listing shows how the function makes use of the preservation criteria to proof that statement coverage is preserved. The tool walks through each non-conditional statement node $B(P_2)$ of the transformed code. Using the continuity relation together with the additional subset relations on the valuation sets it determines those valuation sets that are a subset of valuation set $IV_R(x)$ of the currently investigated node x. Finally, it searches for nodes in the original code that fulfil the coverage-preservation condition for SC. In the first case this happens with node 3 and its 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 prints the function result which is TRUE in this case, meaning that statement coverage has been successfully proved. The functions result-value can be used as one value to be included into the coverage profile.

Statement Coverage is preserved, because of the main property of the transformation. $\delta_4 = \varrho_4$ implies $IV_R(14) \supseteq IV_R(4)$ and $\delta_5 \cup \delta_6 = \varrho_5 \cup \varrho_6$ implies $IV_R(15) \supseteq IV_R(5)$.

Condition Coverage is not preserved, because due to the condition swap $\varrho_4 \subseteq \delta_2$ and no concrete relation can be given between ϱ_2 and δ_2. So the condition $IV_T(13) \supseteq IV_T(c_i)$ for some $c_i \in \{2, 3\}$ cannot be proven.

Decision Coverage is preserved, because of the main property of the transformation.

Modified Condition/Decision Coverage is not preserved, since condition coverage is not preserved.

Path Coverage is not preserved, because for the same reasons as for condition coverage the preservation condition cannot be proven for some paths involving node 13.

6.2. Loop Peeling

Loop peeling is a transformation used to resolve dependences inside a loop, if the source of dependence is a restricted number of iterations like in the example in Figure 12.a. Peeling k iterations from the beginning of a loop means replacing the first k iterations by k copies of the body plus the increment and test code for the loop index variable. The peeled-out code is placed immediately ahead of the loop [2]. Loop peeling can also involve iterations other than the first and the last. In this case the loop must be separated first across the iteration causing the dependence [24].

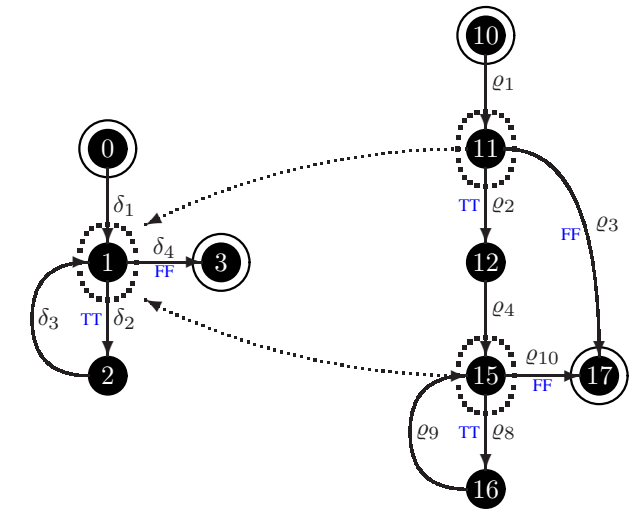
The aCFG in Figure 12.b implements the general case of a loop transformed by peeling out the first iteration ($k = 1$) of the loop. The termination check is assumed to be part of node 1 in the original program at the entry of the loop. In addition to the loop-decision 15 a branch 11 is placed in front of

```
loop i ← 1 to N
   A(i) ← A(i) + A(1);
endloop
```

⇒

```
A(1) ← A(1) + A(1);
loop i ← 2 to N
   A(i) ← A(i) + A(1);
endloop
```

a) Example for application of loop peeling, taken from [24]. The computation uses the value `A(1)` computed in the first iteration. Peeling out the first iteration produces a loop without dependences that can be directly vectorized.



b) aCFGs of a loop peeling transformation with k=1 and a loop decision comprising a single condition.

Figure 12. Transformation: LOOP PEELING

the copy of the loop body of the transformed program to avoid entering the loop, if the loop condition is not fulfilled at the beginning. The statement dealing with the iteration variable is assumed to be part of the loop body.

From point of view of code coverage analysis, this little change in code structure has severe effects on preservation of all coverage criteria. This fact can be considered by an example. To keep the semantics of the loop, the transformation has to fulfil the requirement that each set of input data triggers the same number of executions of the loop body in the original and the transformed version. In the original program, statement coverage, for instance, can be achieved by executing only the first iteration of the loop. Now assume a set of test data, iterating the loop of the original version of the program exactly once. Using the same test data set for the transformed versions only triggers the execution of the first copy of the loop body (node 12 in the aCFG in Figure 12.b). The modified loop will not be entered, because this would cause a second execution of the loop body, violating the basic requirement of the transformation. Thus, the second copy of the loop body will never be triggered by the same set of test-data, and the transformed program will fail to achieve statement coverage. The same applies to other kinds of structural-code coverage.

The requirement, that the number of executions of the loop body has to be invariant with respect to the transformation, implies the equality $\varrho_2 = \delta_2$. Further considerations reveal the relation $\varrho_8 \subseteq \delta_2$, because the transformed loop is not entered for the first iteration. The relations $\varrho_3 \subseteq \delta_4$ and $\varrho_{10} \subseteq \delta_4$ are a consequence of the basic properties derived from the continuity property Equation 11. All these relations are found using the automatic determination functions for transformation relations.

Statement Coverage is not preserved. The preservation condition is not fulfilled for node 16, since $IV_R(16) = \varrho_8 \not\supseteq \delta_2 = IV_R(2)$.

Condition Coverage is not preserved. Since $\varrho_8 \subseteq \delta_2$ the preservation criteria $IV_T(15) \supseteq IV_T(2)$ cannot be proven for condition 15.

Decision Coverage is not preserved for the same reason as condition coverage.

Modified Condition/Decision Coverage not preserved, because neither condition coverage nor decision coverage is preserved.

Path Coverage is not preserved, because the preservation condition is not fulfilled for paths including node 16.

6.3. Loop Inversion

Loop Inversion, in source-language terms, transforms a `while`-loop with the loop-closing test at the begin of the loop into a `repeat/until` loop with the loop-closing test at the end of the loop [2]. In the simplest case it is save to execute the loop body at least once and no additional test is needed on front of the transformed loop. Otherwise a branch decision is generated in front of the loop checking the exit condition to avoid entering the loop if the loop condition is already *false* at the begin. This second case is illustrated in the code example in Figure 13a).

The aCFGs for the general case of the loop-inversion transformation are presented in Figure 13b). In the transformed version, decision $\langle 12 \rangle$ is a conditional branch, which is a modified copy of the original loop decision. It protects the loop-body from execution, if the loop condition is FALSE from the beginning. In addition, the original loop decision $\langle 2 \rangle$ is moved behind the loop body, acting as transformed loop-decision $\langle 14 \rangle$.

The requirement to preserve the semantics of the loop, implies the equality $\delta_2 = \varrho_2$ and $\delta_2 = \varrho_4$ respectively. Informally spoken, this formal conditions means that the loop body of the transformed program must be executed the same number of times for the same input-valuations as in the original program. Splitting the loops-closing decision $\langle 2 \rangle$ into two decisions $\langle 12 \rangle$ and $\langle 14 \rangle$ causes a split of the input-valuation set δ_4 into $\varrho_3 \cup \varrho_6$.

Statement Coverage is preserved, because $\varrho_2 = \delta_2$ and therefore the preservation condition is fulfilled for node 13.

Condition Coverage is not preserved, because of the split $\delta_4 = \varrho_3 \cup \varrho_6$. Therefore $\delta_4 \supseteq \varrho_3$ and $\delta_4 \supseteq \varrho_6$. So, the preservation condition cannot be proved.

Decision Coverage is not preserved for the same principle reason as described for condition coverage.

Modified Condition/Decision Coverage is not preserved, because neither condition coverage nor decision coverage is preserved.

Path Coverage is not preserved. Paths including node 13 cannot fulfill the preservation condition for a similar reason as described for condition coverage.

6.4. Condition Reordering without Short-Circuit Evaluation

The example in this section goes back to the condition reordering problem presented in Section 6.1. It 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, MCDC and PC are now preserved. The main difference here is, that each condition can always decides on the full set of input valuations, because $\delta_2 \cup \delta_3 = \varrho_2 \cup \varrho_3$ is true. Of course, the distribution between δ_2, δ_3 on one side and ϱ_2, ϱ_3 on the other side may differ.

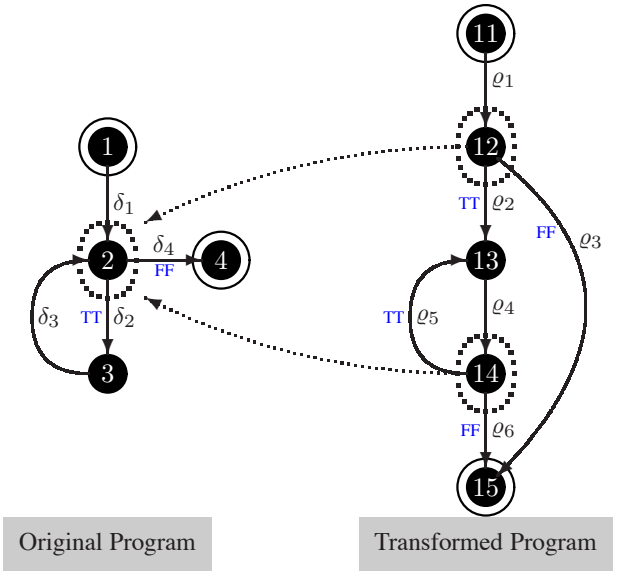
The implementation in Figure 14 implements a fully evaluated branch decision with two conditions connected by logical AND corresponding to the code example given in Figure 10a.

```
i ← 1;
while i ≤ N
    loop_body;
endwhile
```

⇒

```
i ← 1;
if (i ≤ N) then
    do
        loop_body;
    until (i > N);
endif;
```

a) Code example of moving the end test of a loop from before the loop to the end. The missing check of the exit condition in front of the loop is performed with additional code.



b) aCFGs for a loop inversion transformation with an additional loops-closing test placed in front of the loop.

Figure 13. Transformation: LOOP INVERSION

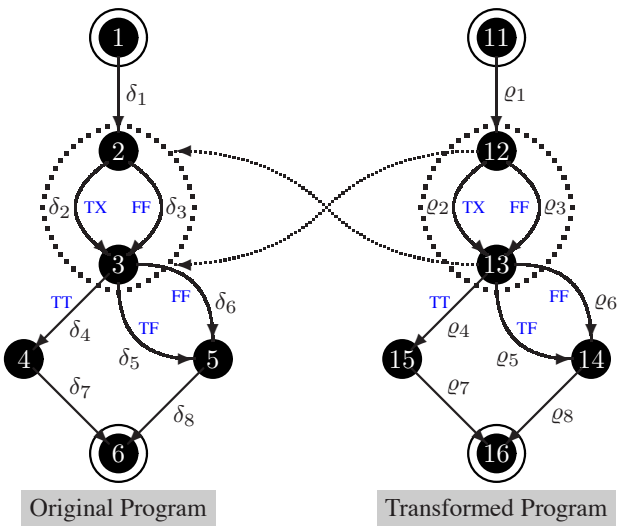


Figure 14. CONDITION REORDERING: Swapping conditions in a branch decision without short-cut semantics.

Since the transformation just swaps the two conditions by leaving the evaluation semantic of the decision unchanged, the input valuations entering the *then* and *else* branch must be kept unchanged. This implies the equality $\delta_4 = \varrho_4$ for the *then* branch and $\delta_5 \cup \delta_6 = \varrho_5 \cup \varrho_6$ for the *else* branch.

The full-evaluated branch statement preserves all kinds of structural-code coverage, because its conditions always can decide on the full input-valuation sets entering the decision. Therefore the produced input-valuation distribution of the conditions is exactly the same. In contrast to the shortcut evaluation, there are no alternative paths where control flow can bypass some conditions.

Statement Coverage is preserved. This is implied by the equalities $\delta_4 = \varrho_4$ and $\delta_5 \cup \delta_6 = \varrho_5 \cup \varrho_6$.

Condition Coverage is preserved, since $\varrho_2 \cup \varrho_3 = \varrho_1 = \delta_1$. Therefore $\varrho_4 = \delta_2$ and $\varrho_5 \cup \varrho_6 = \delta_3$ implies the preservation condition for node 13. The same is true for node 12.

Decision Coverage is preserved, because of the reasons described for condition coverage and decision coverage.

Path Coverage is preserved, because of the equalities described above.

6.5. Strip Mining

Strip mining transforms a single loop into a nested loop operating on strips of the original one. The outer loop steps through the iteration space in blocks of equal size, while the inner loop executes each block. The optimization is used for memory management, for example, to fit the block size handled in the inner loop to the characteristics of the machine [28, 31]. Another application is distributing loop iterations for parallel processing. Strip mining adapts the code to the number of available processors, or justifies iteration blocks for scheduling when single iterations produce not enough work for efficient parallel execution [24]. A possible example for strip-mining the iterations of a for-loop is given in Figure 15a).

The aCFG representations for automatic analysis are presented in Figure 15b). The loop decision $\langle 2, 3\rangle$ is composed of two conditions 2 and 3. The decision of the original program is transformed to two decisions: $\langle 12, 13\rangle$ for the outer loop and $\langle 14, 15\rangle$ for the inner loop. The conditions of decision $\langle 12, 13\rangle$ are derived from the conditions of the original decision $\langle 2, 3\rangle$. However, the two conditions in decision $\langle 14, 15\rangle$ are independent of the conditions in the original decision, as they represent the boundary check and the exit test of the inner loop.

Like most loop transformations, the transformation condition for strip mining requires equality for the number of iterations of the loop body for the same input. This implies the equality $\delta_4 = \varrho_9$ as the essential transformation relation. In addition, the transformation relation $\delta_4 \subseteq \varrho_4$ is true for entering the body of the outer loop, because all execution paths entering the inner loop have to pass this edge. In addition, some execution paths not entering the inner loop may pass this edge, which is the reason that the relation is not an equality.

Most of the essential transformation relations can be obtained systematically without utility functions. On the other hand, the relation $\delta_4 = \varrho_9$ is not determined completely and must be added manually.

Statement Coverage is implied by $\delta_4 = \varrho_9$.

Condition Coverage is not preserved. The preservation condition cannot be proved for conditions 14 and 15, because $\delta_2 \cup \delta_3 = \varrho_2 \cup \varrho_3 \supseteq \varrho_4 = \varrho_7 \cup \varrho_8$.

Decision Coverage is preserved, because of the following relations: $\delta_4 = \varrho_9$, $\delta_4 \subseteq \varrho_{10} \cup \varrho_{11}$ for decision $\{14, 15\}$, and $\delta_4 \subseteq \varrho_4$, $\delta_5 \cup \delta_6 = \varrho_5 \cup \varrho_6$ for decision $\{12, 13\}$.

Modified Condition/Decision Coverage is not preserved because condition coverage is not preserved.

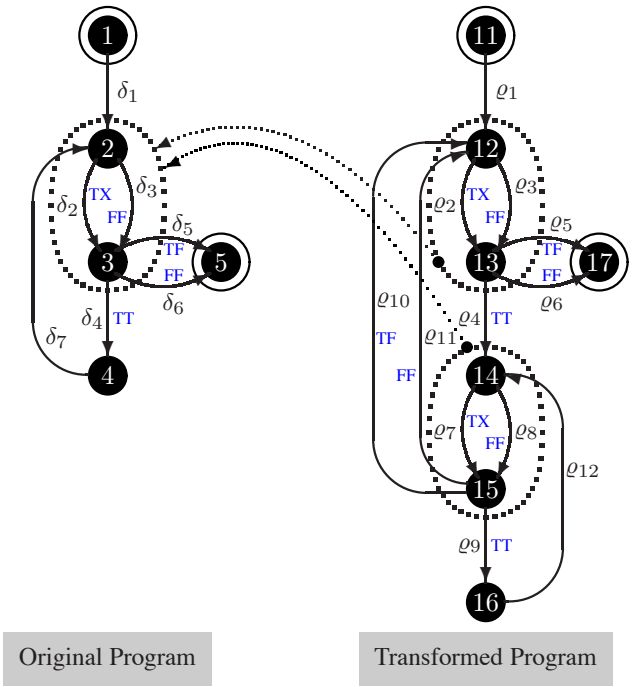
Path Coverage is not preserved for the same principle reason as for condition coverage.

```
for i ← 1 to N do
    loop_body;
endfor;
```

⇒

```
for i ← 1 by 2 to N do
    for i1 ← i to min(i+1,N) do
        loop_body;
    endfor;
endfor;
```

a) Code example for strip mining: Dividing the execution of the left-hand side original loop into strips with a length of 2.



b) aCFGs of Strip Mining: Dividing the iterations of a loop to operate on smaller strips of the original loop.

Figure 15. Transformation: STRIP MINING

6.6. *Branch Optimization*

The purpose of this transformation is to eliminate unnecessary branches by reordering the code and changing branch targets. Branches to branch instructions are often a result of a simple-minded code-generation strategy [32, 2].

The prerequisite situation for branch optimization is comparable with useless code elimination, especially when the branch instruction to remove is a branch to the next instruction. But the situation is a little bit more complicated than it seems on the first view. A possible configuration representative for some types of branch optimization problems is presented in Figure 16. Nodes 2 and 3 are assumed to be unconditional branch statements, symbolized by using circled edges to connect them with their successor. The situation illustrated with the aCFGs is the redirection of the unnecessary branch from node 2 to node 3 directly to the target 4 of the second branch. The problem with this transformation is, as shown on the righthand side of Figure 16, that the redirection possibly leaves the former branch instruction 13 unconnected.

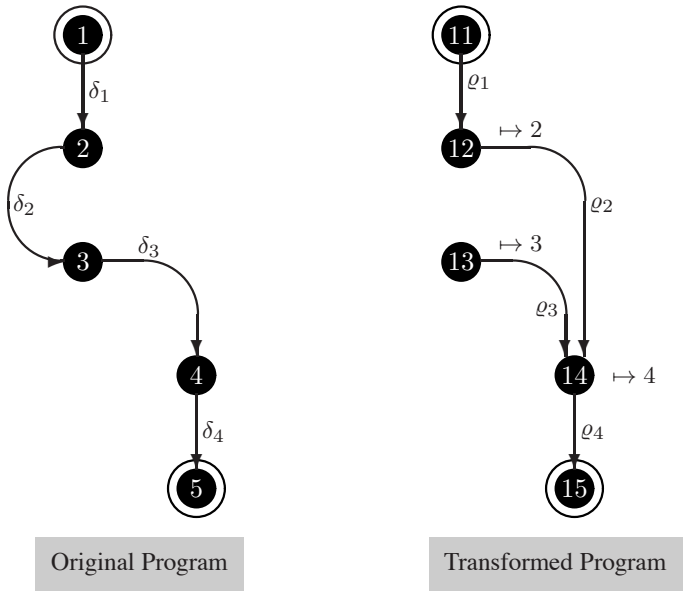
Branch optimization is an example showing some problems concerning our approach to analyze single optimizations. The optimization obviously reduces the reachability of the bypassed branch

```
1: if a ≠ 0 goto 2;
   ⋮
2: if a ≠ 0 goto 3;
   ⋮
3: ...;
```

⇒

```
1: if a ≠ 0 goto 3;
   ⋮
2: if a ≠ 0 goto 3;
   ⋮
3: ...;
```

a) Code example for branch optimization: Passing the second branch is unnecessary and therefore the first branch is redirected. In general, branch 2 cannot be removed afterwards, because other branch statements may use it as a target.



b) aCFGs model for a Branch Optimization.

Figure 16. BRANCH OPTIMIZATION: Redirecting a branch to a branch instruction directly to the branch target.

instruction, and in the worst case it may produce unreachable code. Therefore, statement coverage cannot be achieved in the transformed program. However, optimizations are normally followed by a compiler phase to eliminate dead code. So in the worst case when the branch instruction 13 is completely bypassed, the following dead-code elimination will re-establish statement coverage again.

This use-case also shows some limitations of the chosen program model and its implementation of coverage preservation calculation. The classification TRUE for preservation of coverage criteria other than statement coverage results from the default behavior due to the absence of conditions and decisions in the investigated piece of code. This default behavior seems to makes sense, due to the assumption that the remainder of the program is unchanged and not affected by the code transformation. On the other hand, the implication $DC \Longrightarrow SC$ is obviously not correct. This contradiction is caused by the fact that the investigated program fragment contains no conditions and decisions [13]. But if the bypassed branch will not become a complete orphan, there must be some decision somewhere outside, which addresses this statement, and this decision is now reduced somehow violating the preservation of decision coverage. So the result TRUE for all preservation criteria except statement coverage is caused by the absence of conditions and decisions as well as by the inability due to the restricted view, to decide whether the bypassed instruction is dead code after the transformation or not.

Statement Coverage is not preserved, since $IV_R(13) \subseteq \delta_2 = IV_R(3)$.

Condition Coverage is preserved by default, because the program fragment contains not condition.

Decision Coverage is preserved by default, because the program fragment contains no decision.

Modified Condition/Decision Coverage is preserved by default, because the program fragment contains not conditions and decisions.

Path Coverage is classified as preserved by default. The reason for this behavior is, that it is assumed that node 13 in Figure 16 is either connected by some other path or removed by dead-code elimination. If node 13 is still connected to the remainder of the program, then by the assumption that path coverage is achieved in the original program the path including 13 must be still covered in the transformed program. If node 13 has become an orphan, then dead-code elimination will also remove edge $\langle 13, 14 \rangle$. But then the program fragment will only comprise one path.

6.7. Use-Case Summary

Table I reflects the results of the coverage preservation for the code transformations presented in this section. For the results of *Branch Optimization*, please refer to the remarks given in Section 6.6.

Code Optimization	*Coverage Preservation*				
	SC	*CC*	*DC*	*MCDC*	*PC*
Cond. Reord. (short-circuit)	✓	·	✓	·	·
Loop Peeling	·	·	·	·	·
Loop Inversion	✓	·	·	·	·
Cond. Reord. (non short-circuit)	✓	✓	✓	✓	✓
Strip Mining	✓	·	✓	·	·
Branch Optimization	·	✓	✓	✓	✓

Table I. Calculated Coverage Profiles

7. COMPILER INTEGRATION

In the following we describe in more detail how the integration of the coverage profiles into a compiler has been done and show some performance figures.

7.1. Implementation of the SCCP Mode

The calculation of the coverage profiles is described in Section 5. The principle of how to apply these coverage profiles to a compiler has been already shown in Figure 2.

The following briefly describes how the coverage profiles have been integrated into the open-source compiler GCC 4.5.2 [33, 34]. We have chosen GCC due to its mature code base and the availability to numerous platforms, even processors mainly used in embedded applications. GCC is implemented in ISO C with some additional configuration files from which code is generated automatically.

The following options have been added to the GCC option interface:

--sccp-enforce = metrics-list This option specifies the activation of the SCCP (structural code coverage preservation) mode, which causes the compiler to focus on the coverage preservation of the coverage metrics given as a comma-separated list. Currently, this list can include the items `sc` (statement coverage), `cc` (condition coverage), `dc` (decision coverage), `mcdc` (modified condition-decision coverage), and `pc` (path coverage). For example, to tell the compiler to focus on the preservation of CC and DC, it has to be called with the following option: `--sccp-enforce=cc,dc`.

--sccp-warn When the option `--sccp-enforce=`⟨`metrics-list`⟩ is given to the compiler, the default mode is that the compiler disables all those code optimizations that have the potential to disrupt the coverage metrics specified in ⟨`metrics-list`⟩. With the option `--sccp-warn`

```
struct sccp_coverage_profile_type
{
  enum sccp_code_trans trans; /* id of code optimization */
  unsigned int preserves_mask; /* each bit set represents a preserved metrics */
  int *pflag;                  /* address of enabling flag of optimization */
  const char *trname;          /* name of code optimization */
};

extern struct sccp_coverage_profile_type sccp_coverage_profile[];
```

Figure 17. Source Code of Data Structure of Coverage Profiles

the compiler is meant to be switched to a passive coverage preservation mode, where no code optimizations are guarded, but instead warnings are emitted, which inform the user that the specified coverage metrics may have been disrupted.

At the current stage of implementation the functionality behind the option `--sccp-warn` has not been fully implemented, as experiments so far haven't shown real need for it. The command line interface of GCC has been extended by adding the additional options into the file *common.opt*, from where an existing script of GCC automatically generates the code for parsing and basic handling of the options [34].

The coverage profiles have been added to the files *opts.h* and *opts.c*. In GCC all code optimizations are already guarded by a control flag. For example, the variable `flag_unroll_loops` controls whether the loop unrolling phase is called during compilation or not. GCC provides the flexibility of enabling or disabling each of its supported code optimizations separately by an option, e.g., `--unroll-all-loops` enables loop unrolling. For convenience, there are also generic options to specify the optimizations level of the form `-Ox`) and based on the optimization level `x` a specific subset of the available code optimizations is enabled via their individual control variables. In *opts.h* we have declared the data structure that stores all the coverage profiles. Figure 17 shows the source code of this data structure. `trans` is an integer value that specifies the numerical identifier of each code transformation. This list of identifiers for all supported code optimizations has been added as an enumeration type in ISO C. The variable `preserves_mask` is a bitmask where a one at a bit position indicates that the corresponding coverage metric is preserved by the code transformation. `pflag` is a pointer to the control variable of the optimization; it is used to disable code optimizations in case that they do not preserve the requested coverage metrics. The entry `trname` holds the name of the optimization, which is used for optimization logs.

The initialization of the vector of coverage profiles is done in the source file *opts.c* (variable `sccp_coverage_profile[]` in Figure 17). The coverage profiles have been calculated according to the method described in Section 5. The resulting coverage profiles for GCC 4.5.2 are given in Table II. GCC itself has even more optimizations as those listed in this table. However, our approach was to focus on those optimizations that are implicitly enabled or disabled by the generic optimization option `-Ox`. The optimization level where the individual optimizations gets enabled or disabled are shown in column *Activation*. Furthermore, as shown in row 60-62, we have added support for some loop optimizations which are not implicitly controlled by the option `-Ox`. The general code optimization name each concrete code transformation belongs to is given in Column *Optimization Name*. Coverage profiles for optimizations at function level, e.g., *function inlining*, apply to both subroutines and methods of classes. Further description of the individual code transformations listed in Table II can be found in the GCC documentation [33] using the option name given in column *Command Line Option*. The source code of the prototype implementation of the SCCP compiler can be downloaded from the homepage of the SECCO project [35].

What we already see from Table II is that most of the studied code optimizations of GCC do preserve the individual structural code-coverage metrics. However, an empirical evaluation is necessary to see quantitative values of how the activation of coverage preservation will have an impact on the performance of the generated code.

No	*Optimization Name*	*Command Line Option*	*Activation*	*SC*	*CC*	*DC*	*MCDC*	*PC*
1	Align Branch Target Address	falign-functions	$\leq$ O1	✓	✓	✓	✓	✓
2	Align Branch Target Address	falign-jumps	$\leq$ O1	✓	✓	✓	✓	✓
3	Align Branch Target Address	falign-labels	$\leq$ O1	✓	✓	✓	✓	✓
4	Align Branch Target Address	falign-loops	$\leq$ O1	✓	✓	✓	✓	✓
5	Copy Propagation	fcprop-registers	$\geq$ O1	✓	✓	✓	✓	✓
6	Defer Arg Pop Until Later	fdefer-pop	$\geq$ O1	✓	✓	✓	✓	✓
7	Branch Delay Slot Filling	fdelayed-branch	$\geq$ O1	✓	✓	✓	✓	✓
8	Forward Propagation	fforward-propagate	$\geq$ O1	✓	✓	✓	✓	✓
9	Guess Branch Probabilities	fguess-branch-probability	$\geq$ O1	✓	✓	✓	✓	✓
10	If Conversion	fif-conversion	$\geq$ O1	✓	✓	✓	✓	✓
11	If Conversion	fif-conversion2	$\geq$ O1	✓	✓	✓	✓	✓
12	Detection of Pure/Const Functions	fipa-pure-const	$\geq$ O1	✓	✓	✓	✓	✓
13	Detect local-only references	fipa-reference	$\geq$ O1	✓	✓	✓	✓	✓
14	Constant Merging	fmerge-all-constants	$\geq$ O1	✓	✓	✓	✓	✓
15	Constant Merging	fmerge-constants	$\geq$ O1	✓	✓	✓	✓	✓
16	Stack Frame Creation Avoidance	fomit-frame-pointer	$\geq$ O1	✓	✓	✓	✓	✓
17	Split Wide Types	fsplit-wide-types	$\geq$ O1	✓	✓	✓	✓	✓
18	Constant Propagation (Conditional)	ftree-ccp	$\geq$ O1	✓	✓	✓	✓	✓
19	Loop Inversion	ftree-ch	$\geq$ O1	✓	✓	✹	✹	✹
20	Variable Renaming	ftree-copyrename	$\geq$ O1	✓	✓	✓	✓	✓
21	Copy Propagation	ftree-copy-prop	$\geq$ O1	✓	✓	✓	✓	✓
22	Useless Code Elimination	ftree-dce	$\geq$ O1	✓	✓	✓	✓	✓
23	Diverse (incl. Jump Threading)	ftree-dominator-opts	$\geq$ O1	✹	✹	✹	✹	✹
24	Useless Code Elimination	ftree-dse	$\geq$ O1	✓	✓	✓	✓	✓
25	Redundancy Elimination (full only)	ftree-fre	$\geq$ O1	✓	✓	✓	✓	✓
26	Forward Store Motion	ftree-sink	$\geq$ O1	✓	✓	✓	✓	✓
27	Scalar Replacement of Aggregates	ftree-sra	$\geq$ O1	✓	✓	✓	✓	✓
28	Copy Propagation	ftree-ter	$\geq$ O1	✓	✓	✓	✓	✓
29	Caller Register Optimization	fcaller-saves	$\geq$ O2	✓	✓	✓	✓	✓
30	Tail Merging (aka Cross Jumping)	fcrossjumping	$\geq$ O2	✓	✓	✓	✓	✓
31	CSE (across BB)	fcse-follow-jumps	$\geq$ O2	✓	✓	✓	✓	✓
32	Scalar Replacement of Aggregates	fipa-sra	$\geq$ O2	✓	✓	✓	✓	✓
33	Influences Other Optimizations	fexpensive-optimizations	$\geq$ O2	✓	✓	✓	✓	✓
34	Global CSE	fgcse	$\geq$ O2	✓	✓	✓	✓	✓
35	Indirect Function Inlining	findirect-inlining	$\geq$ O2	✹	✹	✹	✹	✹
36	Function Inlining	finline-small-functions	$\geq$ O2	✹	✹	✹	✹	✹
37	Constant Propagation (Interproc.)	fipa-cp	$\geq$ O2	✓	✓	✓	✓	✓
38	Reassign Reg-NRs in Moves	foptimize-register-move	$\geq$ O2	✓	✓	✓	✓	✓
39	Optimize sibling/tail-recursive calls	foptimize-sibling-calls	$\geq$ O2	✓	✓	✓	✓	✓
40	Peephole Optimization	fpeephole2	$\geq$ O2	✓	✓	✓	✓	✓
41	Register Move	fregmove	$\geq$ O2	✓	✓	✓	✓	✓
42	Reorder Basic Blocks	freorder-blocks	$\geq$ O2	✓	✓	✓	✓	✓
43	Reorder Functions	freorder-functions	$\geq$ O2	✓	✓	✓	✓	✓
44	CSE	frerun-cse-after-loop	$\geq$ O2	✓	✓	✓	✓	✓
45	Instruction Reordering	fschedule-insns	$\geq$ O2	✓	✓	✓	✓	✓
46	Instruction Reordering	fschedule-insns2	$\geq$ O2	✓	✓	✓	✓	✓
47	Assume Strict Aliasing Rules	fstrict-aliasing	$\geq$ O2	✹	✹	✹	✹	✹
48	Assume No Overflow	fstrict-overflow	$\geq$ O2	✹	✹	✹	✹	✹
49	Bypass Subsumed Cond. Branches	fthread-jumps	$\geq$ O2	✹	✹	✹	✹	✹
50	Switch Conversion	ftree-switch-conversion	$\geq$ O2	✓	✓	✓	✓	✓
51	Partial Redundancy Elimination	ftree-pre	$\geq$ O2	✓	✓	✓	✓	✓
52	Value Range Propagation	ftree-vrp	$\geq$ O2	✓	✓	✓	✓	✓
53	Useless Code Elimination	ftree-builtin-call-dce	$\geq$ O2	✓	✓	✓	✓	✓
54	CSE	fgcse-after-reload	$\geq$ O3	✓	✓	✓	✓	✓
55	Function Inlining	finline-functions	$\geq$ O3	✹	✹	✹	✹	✹
56	Function Cloning	fipa-cp-clone	$\geq$ O3	✹	✹	✹	✹	✹
57	Predictive Commoning Optimization	fpredictive-commoning	$\geq$ O3	✓	✓	✓	✓	✓
58	Loop Unswitching	funswitch-loops	$\geq$ O3	✓	✹	✓	✹	✹
59	Loop Vectorization	ftree-vectorize	$\geq$ O3	✓	✓	✓	✓	✓
60	Loop Unrolling	funroll-loops	explicit	✹	✹	✹	✹	✹
61	Loop Unrolling	funroll-all-loops	explicit	✹	✹	✹	✹	✹
62	No Loop Var Overflow	funsafe-loop-optimizations	explicit	✹	✹	✹	✹	✹

Legend: ✓...coverage preserved ✹...coverage not preserved

Table II. Coverage Preservation Profile for GCC (version 4.5.2)

In this section we presented the basic mechanism of how we implemented the SCCP profiles in GCC. For other compilers we expect that the integration is of similar ease. However, what made the implementation in GCC nice to test is the ability to control the activation of any code optimization individually. The source code of the prototype implementation of the SCCP compiler can be downloaded from the homepage of the SECCO project [35].

7.2. Evaluation

Table IV shows some quantitative numbers on the performance impact of the SCCP mode as implemented in GCC 4.5.2. The first benchmark *coremark* is a synthetic benchmark suite from the EEMBC organization, designed to measure the performance CPUs of embedded systems [36]. The other benchmarks are taken from the standard worst-case execution time (WCET) benchmark suite assembled by the Mälardalen University [37]. The Mälardalen WCET benchmarks have been slightly modified in order to show the effects of different optimization levels. This modification consists of adding a new entry routine that calls the original entry routine several times. With this modification the program execution times become a more dominant timing contribution compared to the program call overhead. The *core mark* benchmark is relatively large, aiming to show an average performance impact by coverage preservation. The Mälardalen WCET benchmarks are relatively small with distinct control-flow structures, having been selected to query the performance-impact range of coverage preservation.

The experiments were run on a Mac OS-X 10.6 machine with an Intel Core 2 Duo processor running at 3.06 MHz. The execution time measurements where done for different optimization settings of the compiler, using a measurement accuracy of 1ms. The different compiler options are specified in Table III. With the setting *SCCP1* we test the performance impact in case of enforcing only the most simple structural code coverage, i.e., statement coverage. In contrast, with *SCCP2* we test the performance impact in case of enforcing all supported coverage metrics.

Setting	*Command-line Option for GCC*
O0	-O0
O1	-O1
O2	-O2
O3	-O3
FULL	-O3 –unroll-loops –unroll-all-loops
SCCP1	-O3 –sccp-enforce –sccp=sc
SCCP2	-O3 –sccp-enforce –sccp=sc,cc,dc,mcdc,pc

Table III. Compiler Options for the Different Settings

From Table IV we actually see that the SCCP mode does quite well with respect to code performance. First of all, we also see that it is not always the maximum optimization level that leads to the best performance. For example, in case of benchmark *insertsort* the optimization level `O2` shows a better performance that `O3`. This can have different causes, e.g., a higher optimization level causes an increase in code size, which subsequently can lead to a less performant code. As the setting of *SCCP1* and *SCCP2* is based on the compiler option `-O3`, the most useful result is their comparison to the setting *O3*. The setting *FULL* includes further loop optimizations that are not always beneficial, but are interesting for comparison.

We have to note that some measurements for the benchmarks *expint*, *fir*, and *fft1* are close to the measurement accuracy, thus the relative comparison of the different settings does not make that much sense for these benchmarks. However, these benchmarks still demonstrate that the SCCP modes are competitive in performance, allowing for the same highly effective optimizations as in the settings *O2*, *O3*, and *FULL*.

The execution time results of Table IV are summarized in Figure 18. The figure shows for each benchmark the relative performance for the different compilation settings. A performance of 100% of a compilation setting means that the execution time for this setting was minimal. As said above,

		Measured Execution Time [ms]						
Name	*#LOC*	*O0*	*O1*	*O2*	*O3*	*FULL*	*SCCP1*	*SCCP2*
coremark	5569	$109.0_{\cdot 10^3}$	$48.4_{\cdot 10^3}$	$43.5_{\cdot 10^3}$	$35.2_{\cdot 10^3}$	$36.0_{\cdot 10^3}$	$35.9_{\cdot 10^3}$	$35.7_{\cdot 10^3}$
statemate	1276	228.00	187.00	260.00	292.00	261.00	284.00	285.00
insertsort	91	48.00	33.00	21.00	25.00	27.00	28.00	27.00
compress	507	150.00	102.00	96.00	91.00	85.00	91.00	91.00
nsichneu	4253	128.00	45.00	38.00	40.00	39.00	40.00	40.00
qurt	166	45.00	27.00	29.00	28.00	24.00	29.00	28.00
adpcm	878	99.00	55.00	71.00	65.00	64.00	65.00	65.00
matmult	177	53.00	25.00	22.00	15.00	13.00	15.00	15.00
ludcmp	147	110.00	46.00	42.00	34.00	40.00	34.00	34.00
jfdctint	375	77.00	35.00	34.00	32.00	31.00	32.00	31.00
crc	128	31.00	19.00	20.00	18.00	18.00	18.00	18.00
edn	285	128.00	43.00	42.00	29.00	29.00	29.00	28.00
expint	157	1746.00	1399.00	2.00	3.00	2.00	3.00	2.00
fdct	239	74.00	27.00	28.00	20.00	19.00	20.00	20.00
fir	276	1006.00	348.00	347.00	2.00	2.00	3.00	2.00
fft1	219	63.00	19.00	12.00	2.00	3.00	3.00	3.00
ndes	231	171.00	106.00	54.00	37.00	38.00	37.00	37.00

Table IV. Performance evaluation of SCCP

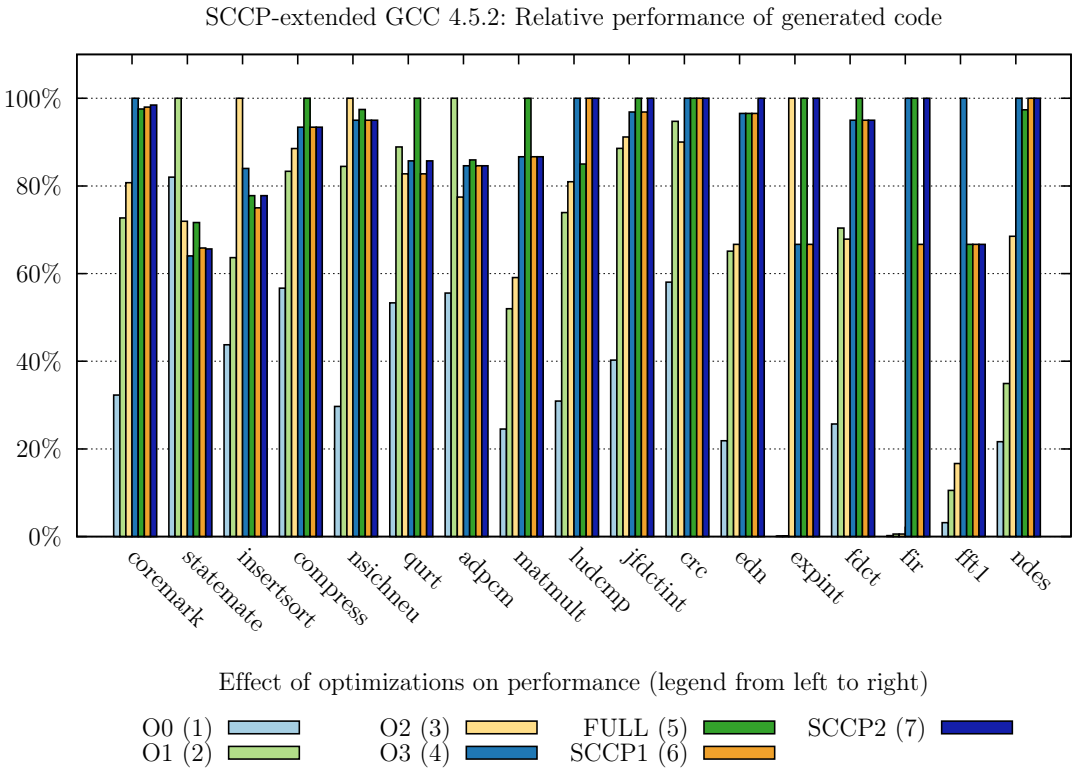


Figure 18. Performance comparison: the performance of each optimization opt_i is normalized to the minimal performance of all optimizations: $\frac{\min_j time(opt_j)}{time(opt_i)}$

the results of *expint*, *fir*, and *fft1* are close to the measurement granularity and thus their relative presentation in Figure 18 has to be taken with care. However, from the remaining benchmarks we already see good evidence that the activation of the SCCP mode does not have a significant performance influence.

This leads to the conclusion that the gain in improved correctness trust of programs by having ensured at machine-code level the same structural code coverage as at source-code level comes with a low and mostly even negligible cost. The impact of this quite positive result can be quite manifold. For our original motivation of measurement-based timing analysis [6, 10] this means that systematic generation of test data at source-code level will become robust with respect to control-flow coverage. The results also have a quite positive message for functional testing of software. For example in embedded computing where performance based on code optimizations is highly important due to the limitation of computing resources, the SCCP provides better trust in program correctness without sacrificing processor utilization or battery lifetime.

8. SUMMARY AND CONCLUSION

In this article 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), decision coverage (DC), modified condition-decision coverage (MCDC), and (scoped) path coverage (PC).

The implementation of the coverage preservation mode into the GCC 4.5.2 compiler showed that the preservation of structural code coverage in GCC comes at quite low cost. In most cases it was as good as normal compilation with optimization flag `-O3`, and in cases where the additional loop optimizations proved to be effective, the coverage preservation stayed slightly behind. In conclusion we have shown that structural coverage preservation can be integrated into a compiler with quite reasonable effort and we hope that this technique will be adopted by compiler vendors as well, especially in the domain of embedded computing, where thorough testing of functional and extra-functional behavior is essential.

REFERENCES

1. RTCA. Software considerations in airborne systems and equipment certification. RTCA/DO-178B 1992.
2. Muchnick SS. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN: 1-55860-320-4.
3. Kirner R. SCCP/x - a compilation profile to support testing and verification of optimized code. *Proc. ACM Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*, Salzburg, Austria, 2007; 38–42.
4. Kirner R. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems* 2009; **2009**, doi:10.1155/2009/127945.
5. Jaramillo C, Gupta R, Soffa ML. Debugging and testing optimizers through comparison checking. *Electronic Notes in Theoretical Computer Science* Apr 2002; **65**(2):83–99. COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002).
6. Wenzel I, Kirner R, Rieder B, Puschner P. Measurement-based timing analysis. *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, 2008.
7. Kirner R, Puschner P, Wenzel I. Measurement-based worst-case execution time analysis using automatic test-data generation. *Proc. 4th International Workshop on Worst-Case Execution Time Analysis*, Catania, Italy, 2004; 67–70.
8. Aho AV, Sethi R, Ullman JD. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1997. ISBN: 0-201-10088-6.

9. Wenzel I, Rieder B, Kirner R, Puschner P. Automatic timing model generation by CFG partitioning and model checking. *Proc. Conference on Design, Automation and Test in Europe (DATE'05)*, IEEE: Munich, Germany, 2005; 606–611.
10. Bünte S, Zolda M, Tautschnig M, Kirner R. Improving the confidence in measurement-based timing analysis. *Proc. 14th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Newport Beach, CA, USA, 2011.
11. Chilenski JJ. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. *Technical Report DOT/FAA/AR-01/18*, Boeing Commercial Airplane Group Apr 2001.
12. ISO. *Programming Languages – C*. 2nd edn., ISO/IEC 9899:1999, 1999. Technical Committe: JTC 1/SC 22/WG 14.
13. Myers GJ. *The Art of Software Testing*. John Wiley & Sons, 1979.
14. Hayhurst KJ, Veerhusen DS, Chilenski JJ, Rierson LK. A practical tutoral on modified condition/decision coverage. *Technical Report NASA/TM-2001-210876*, National Aeronautics and Space Administration, Hampton, Virginia May 2001. Available in pdf format.
15. Bhansali PV. The mcdc paradox. *SIGSOFT Software Engineering Notes* May 2007; **32**(3):1–4, doi:http://doi.acm.org/10.1145/1241572.1241581.
16. Kandl S, Kirner R. Error detection rate of MC/DC for a case study from the automotive domain. *Proc. 8th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'10)*, Waidhofen/Ybbs, Austria, 2010.
17. Allen FE. Control flow analysis. *SIGPLAN Not.* 1970; **5**(7):1–19, doi:http://doi.acm.org/10.1145/390013.808479.
18. Shivers O. Control-flow analysis in scheme. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988; 164–174, doi:http://doi.acm.org/10.1145/53990.54007.
19. Cota BA, Fritz DG, Sargent RG. Control flow graphs as a representation language. *WSC '94: Proceedings of the 26th conference on Winter simulation*, Society for Computer Simulation International: San Diego, CA, USA, 1994; 555–559.
20. Heintze N. Control-flow analysis and type systems. *Proc. of 2nd International Symposium on Static Analysis*, *LNCS*, vol. 983, Springer: Glasgow, UK, 1995; 189–206.
21. Harel D. On visual formalisms. *Commun. ACM* 1988; **31**(5):514–530, doi:http://doi.acm.org/10.1145/42411.42414.
22. Gross JL, Yellen J. *Graph Theory and Its Applications*. 2nd edn., Chapman and Hall/CRC, 2006.
23. Ahuja RK, Magnanti TL, Orlin JB. *Network Flows: Theory, Algorithms and Applications*. PRENTICE HALL, 1993.
24. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures*. Elsevier Academic Press, 2002.
25. Wolfram S. *The Mathematica Book*. 4th edn., Cambridge University Press, 1999.
26. Haas W. Automatic coverage-profile calculation for code optimization. Master's Thesis, Technische Universität Wien, Vienna, Austria Dec 2010.
27. Skiena SS. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley Publishing Co., 1990.
28. Mercer R. The convex fortran 5.0 compiler. *Proceedings of the 3rd International Conference on Supercomputing, Vol. 2*, 1988.
29. Organisation IS. *ISO/IEC 14882:2003 Programming Languages - C++*. American National Standards Institute: New York, 2003. Technical Committe: JTC 1/SC 22/WG 21.
30. Ada reference manual iso/iec 8652:1995(e) with technical corrigendum 1 language and standard libraries.
31. Padua DA, Wolfe MJ. Advanced compiler optimizations for supercomputers. *Commun. ACM* 1986; **29**(12):1184–1201, doi:http://doi.acm.org/10.1145/7902.7904.
32. Lowry ES, Medlock CW. Object code optimization. *Commun. ACM* 1969; **12**(1):13–22, doi:http://doi.acm.org/10.1145/362835.362838.
33. Stallman RM, GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press, Boston, USA Dec 2010. GCC version 4.5.2, available online at `http://gcc.gnu.org/gcc-4.5/`.
34. Stallman RM, GCC Developer Community. *GNU Compiler Collection Internals*. GNU Press, Boston, USA Dec 2010. GCC version 4.5.2, available online at `http://gcc.gnu.org/gcc-4.5/`.
35. Vienna University of Technology. The SECCO project: Sustaining entire code-coverage on code optimization. web page (`http://pan.vmars.tuwien.ac.at/secco/`). Accessed in May 2011.
36. Coremark v1.0. Web page (`http://www.coremark.org`) 2009. Accessed online in Dez. 2011.
37. Mälardalen research and technology centre WCET benchmarks. Web page (`http://www.mrtc.mdh.se/projects/wcet/`) 2009. Accessed online in April 2011.
38. Whalen M, Rajan A, Heimdahl M, Miller S. Coverage metrics for requirements-based testing. *International Symposium on Software Testing and Analysis*, Portland, USA, 2006.
39. Vilkomir SA, Bowen JP. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. *Formal Aspects of Computing* Mar 2006; **18**(1):42–62. DOI: 10.1007/s00165-005-0084-7.
40. Vilkomir SA, Bowen JP. Formalization of software testing criteria using the z notation. *Proc. 25th Annual International Computer Software and Applications Conference*, Honolulu, Hawaii, USA, 2001; 351.
41. Chilenski JJ, Miller S. Applicability of modified condition decision coverage to software testing. *Software Engineering Journal* Sep 1994; **7**(5).
42. Object Management Group. *MDA Guide*. Version 1.0.1 edn. June 2003. Document number: omg/2003-06-01.
43. Broy M, Jonsson B, Katoen JP, Leucker M, Pretschner A ((eds.)). *Model-based Testing of Reactive Systems*, *Lecture Notes in Computer Science*, vol. 3472. Springer, 2005. ISBN: 3-540-26278-4.
44. Heimdahl MPE, Whalen M, Rajan A, Miller SP. Testing strategies for model-based development. *Technical Report NASA/CR-2006-214307*, National Aeronautics and Space Administration, Hampton, Virginia Apr 2006. Available in pdf format.
45. Rajan A, Whalen M, Heimdahl M. Model validation using automatically generated requirements-based tests. *Proc. 10th IEEE High Assurance Systems Engineering Symposium*, Dallas, USA, 2007.

46. Baresel A, Conrad M, Sadeghipour S, Wegener J. The interplay between model coverage and code coverage. *Proc. of Int'l European Conference on Software Testing, Analysis and Review (EuroSTAR 2003), to appear*, 2003.
47. Rajan A, Whalen M, Heimdahl M. The effect of program and model structure on MC/DC test adequacy coverage. *Proc. 30th Int'l Conference on Software Engineering*, Leipzig, Germany, 2008.
48. Elbaum S, Gable D, Rothermel G. The impact of software evolution on code coverage information. *Proc. International Conference on Software Maintenance*, 2001; 169–179.
49. Harman M, Hu L, Hierons R, Wegener J, Sthamer H, Baresel A, Roper M. Testability transformation. *IEEE Transactions on Software Engineering* Jan 2004; **30**(1).