Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 127945, 16 pages
doi:10.1155/2009/127945

Research Article

Towards Preserving Model Coverage and Structural Code Coverage

Raimund Kirner

Institut für Technische Informatik, Technische Universität Wien, Treitlstraße 3/182/1, A-1040 Wien, Austria

Correspondence should be addressed to Raimund Kirner, raimund@vmars.tuwien.ac.at

Received 12 August 2008; Revised 20 January 2009; Accepted 21 February 2009

Recommended by Bernhard Rinner

Embedded systems are often used in safety-critical environments. Thus, thorough testing of them is mandatory. To achieve a required structural code-coverage criteria it is beneficial to derive the test data at a higher program-representation level than machine code. Higher program-representation levels include, beside the source-code level, languages of domain-specific modeling environments with automatic code generation. For a testing framework with automatic generation of test data this will enable high retargetability of the framework. In this article we address the challenge of ensuring that the structural code coverage achieved at a higher program representation level is preserved during the code generations and code transformations down to machine code. We define the formal properties that have to be fullfilled by a code transformation to guarantee preservation of structural code coverage. Based on these properties we discuss how to preserve code coverage achieved at source-code level. Additionally, we discuss how structural code coverage at model level could be preserved. The results presented in this article are aimed toward the integration of support for preserving structural code coverage into compilers and code generators.

1. Introduction

Testing is a mandatory process to assess the correct behavior of safety-critical systems. Even the increasing use of formal verification cannot make testing obsolete, as there is always a gap between the formal model and the real system with all the issues of integration.

The use of formal (=executable) models increasingly pervades the software engineering process. Formal models are used as part of the specification, as high-level software descriptions with automatic code generation, or as a tool for formal verification and model-based testing [1].

When generating test data it is beneficial to operate at the same representation level where the software is developed, which may be at the source-code level or at a domain-specific modeling environment like ezRealtime [2], MATLAB/Simulink [3, 4], Statemate [5], or Scade [6]. The advantage of test-data generation at this high-level program representation is on the one side reduced complexity and availability of explicit knowledge of the program behavior that might get lost during code generation and compilation. On the other side, a test-data generator operating on such a high-level program representation could be easily retargeted to different platforms. Beside conventional testing, the support for retargetability is also of high interest for hybrid timing analysis, that is, an approach to determine the timing behavior of a program based on the combination of execution-time measurements and program analyses [7, 8].

Structural code-coverage criteria are metrics to analyze and quantify the control-flow coverage that is achieved for a given set of test data. Using a model-based or source-based test-data generator raises the challenge of ensuring that adequate structural code-coverage has been achieved at machine-code level [9]. Code generators and compilers perform many transformations on the program or model given as input. Some of these code transformations can compromise structural code-coverage by copying, reordering, or moving conditions inside the program or even creating new conditions and decisions. For example, optimizations like loop unrolling, loop inversion, reverse if-conversion, and condition reordering [10] can disrupt full structural code-coverage. In general, full structural code-coverage cannot be guaranteed in this case without taking the burden of analyzing the machine code.

We propose an approach toward the preservation of structural code coverage when transforming the program. To

achieve this, we introduce in Section 3 a notation to formally define structural code-coverage criteria. In Section 4 we present coverage preservation criteria for the different variants of structural code coverage. As described in Section 5, these criteria can help to extend a compiler with the ability of preserving coverage achieved at source level. The code coverage is preserved by prohibiting all code transformations that can disrupt the concrete structural code coverage metric. If full coverage preservation is not strictly required, the compiler may be used in a special mode where all available code transformations are allowed but a warning is emitted if structural code coverage may be compromised by an applied code optimization. Issues of preserving model coverage by code generators are discussed in Section 6.

2. Related Work

Structural coverage criteria are used as a supplementary criterion to monitor the progress of testing [11]. The DO178b document introduces the *modified condition-decision coverage* (MCDC) as a supplementary criterion for testing systems of safety-criticality level A [12]. Vilkomir proposes solutions to overcome some weaknesses of MCDC [13]. Vilkomir and Bowen have formally modeled structural code-coverage criteria using the Z notation [14]. The formalization we present in this article is basically equivalent, with the difference that we also support hidden-control flow [15], 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 [16]. 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 [1, 17]. Using models to verify the correctness of the system requires evidence of the model's correctness [18].

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 [19]. 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 [20]. To test safety-critical systems we want to do better than relying on accidental coverage correlations.

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 [21]. Their results also motivate our work for the preservation of code coverage.

A method complementary to our approach is described by Harman et al. Testability transformation results in a transformed program to be used by a test-data generator to improve its ability of generating test data for the original program [22].

3. Basic Definitions

In this section we give a list of basic definitions. These definitions are used to describe properties of structural code coverage and to preserve structural code coverage.

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

Control-Flow Graph (CFG). Is used to model the control flow of a program [23]. A CFG $G = \langle N, E, s, t\rangle$ consists of a set of nodes N representing *basic blocks* (see below), a set of edges $E : N \times N$ representing the control flow (also called control-flow edges), a unique entry node s, and a unique end node t.

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_i as $\mathrm{PS}(P_i)$. The concrete partitioning of a program into scopes is application-specific. For example, in [24] a program partitioning is used that allows to trade the number of required test data against the number of instrumentation points. Another feature of scopes is that nested scopes can be used to hide details. This feature allows to reduce the program complexity of the surrounding scope.

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, that is, the scoped paths where the program scope subsumes the whole program, is denoted as $\mathrm{PP}(P)$.

Basic Block. Of a program P is a code sequence of maximal length with a single entry point at the beginning and with the only allowed occurrence of a control-flow statement at its end. We denote the set of basic blocks in a program P_i as $\mathrm{B}(P_i)$. The set of all basic blocks along a scoped path pp is denoted as $\mathrm{B}(pp)$. Note that in cases of program paths with cycles, $\mathrm{B}(pp)$ will contain multiple instances of the basic blocks in the program code. If a scoped path goes through a nested program scope, all the basic blocks from the nested program scope are hidden for this path. The starting basic block of a scoped path pp is denoted as $\mathrm{B}_S(pp)$.

Decision. Is a Boolean expression composed of *conditions* that are combined by Boolean operators. If a *condition* occurs more than once in the *decision*, each occurrence is a distinct condition [25]. However, the *input* of a decision is the set of its conditions without duplicates. A decision is composed of one or more basic blocks. We denote the set of decisions of a program P_i as $\mathrm{D}(P_i)$.

There are source languages, where decisions are hidden by an implicit control flow. For example, in ANSI C due to

the short-circuit evaluation the following statement `a = (b && c);` contains the decision `(b && c)`. The short-circuit evaluation of ANSI 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. See Section 5.4 for further details. The correct identification of hidden control flow is important, for example, to analyze decision coverage.

Condition. Is a Boolean expression. We consider only lowest-level conditions, that is, conditions that do not contain operators with Boolean arguments [25]. A condition is composed of one or more basic blocks. We denote the set of conditions of a decision d as $\mathrm{C}(d)$. The set of all conditions within a program P_i is denoted as $\mathrm{C}(P_i)$.

The set of all conditions that directly control edges along a scoped path pp is denoted as $C(pp)$. Note that in cases of program paths with cycles, $C(pp)$ will contain multiple instances of the conditions in the program code. If a scoped path goes through a nested program scope, all the conditions from the nested program scope are hidden for this path. To follow a certain path, it is also important whether a condition evaluates to TRUE/FALSE. Whether a condition has to be evaluated as TRUE or as FALSE is given by the syntactical structure of a program. For a given scoped path pp we denote by $C_T(pp)$ all the conditions that have to be evaluated as TRUE and by $C_F(pp)$ all the conditions that have to be evaluated as FALSE to follow pp. It holds that $C_T(pp) \cup C_F(pp) = C(pp)$ and $(C_T(pp) \cap C_F(pp)) = \emptyset$.

Input Data $\mathbb{ID}$. Defines the set of all possible valuations of the input variables of a program. (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.)

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 consider exhaustive testing ($\mathbb{TD} = \mathbb{ID}$) there would be no challenge of structural code-coverage preservation.

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-dataset, where $x$refers to TRUE and $IV_F(x)$ is the set, where $x$refers to FALSE. The following properties always hold for $IV_T(x), IV_F(x)$:

$$\begin{aligned} IV_T(x) \cap IV_F(x) &= \emptyset, \\ IV_T(x) \cup IV_F(x) &= IV_R(x). \end{aligned} \tag{1}$$

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

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

For this code fragment we assume

$$\mathbf{IV_R(a{=}{=}3)} = \{\langle \mathbf{a}, \mathbf{b}\rangle \;\; \mathbf{a}, \mathbf{b} \in \mathbf{int}\}. \tag{2}$$

It follows that

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

(and not the larger set $\{\langle \mathbf{a}, \mathbf{b}\rangle \mid \mathbf{a}, \mathbf{b} \in \mathbf{int}\}$ due to the hidden control flow caused by the short-circuit evaluation of ANSI C; see Section 5.4). It follows that

$$\mathbf{IV_T(b{=}{=}2)} = \{\langle 3, 2\rangle\}. \tag{4}$$

Only those input data that trigger the execution of condition `b==2` and evaluate it to TRUE are within $\mathbf{IV_T}$(`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

$$\mathbf{IV_F(b{=}{=}2)} = \{\langle 3, \mathbf{b}\rangle \mid \mathbf{b} \in \mathbf{int} \wedge \mathbf{b} \neq 2\}. \tag{5}$$

The definition of $\mathbf{IV_R(x)}$, $\mathbf{IV_T(x)}$, and $\mathbf{IV_F(x)}$ depends on whether the programming language has hidden control flow (see Section 5.4). Above definitions allow to formally describe structural code coverage criteria. We will also use them to describe requirements to preserve structural code coverage.

3.1. Structural Code-Coverage Criteria. Structural code-coverage criteria are metrics to analyze and quantify the control-flow coverage that is achieved for a given set of test data. Execution traces are used to collect the coverage information. In general, the satisfaction of a structural code-coverage criterion is not the primary test-case generation strategy in functional testing. Instead, structural code-coverage achieved during testing is analyzed as a supplementary measure to decide whether the implemented functionality has been sufficiently tested and does not contain any unintended functionality. However, there are also rare testing scenarios where the satisfaction of a certain code-coverage is the primary directive for test-data generation. For example, in measurement-based timing analysis an estimation of the worst-case execution time (WCET) is derived by systematic measurements [24].

In the following we review the properties of several structural code-coverage criteria.

Line Coverage. Is not a serious code coverage criterion, as without strict coding guidelines there is an ambiguous mapping from source lines to statements. In the extreme case one could write the whole program within one source line. Historically, line coverage was used as an easy hack when tools for analyzing *statement coverage* were missing. Thus, we do not discuss preservation of line coverage in this work.

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 [26] and should best

be considered as a minimal requirement. Using our above definitions, we can formally define SC as follows:

$$\forall b \in \mathrm{B}(P).\ (\mathbb{TD} \cap IV_R(b)) \neq \emptyset. \tag{6}$$

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

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*. Decision coverage implies *statement coverage:*

$$\forall d \in \mathrm{D}(P).\ (IV_T\ (d) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F\ (d) \cap \mathbb{TD}) \neq \emptyset. \tag{7}$$

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 (8)

$$\forall c \in \mathrm{C}(P).\ (IV_T(c) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F(c) \cap \mathbb{TD}) \neq \emptyset. \tag{8}$$

Note that our definition requires in case of short-circuit operators that each condition is really executed. This is achieved by the semantics of $IV_T()$, $IV_F()$. However, often definitions are used that do not explicitly consider short-circuit operators (e.g., [27]), 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.

Condition/Decision Coverage (CDC). Requires that both, *condition coverage* and *decision coverage* are achieved.

Modified Condition/Decision Coverage (MCDC). Requires to show that each condition can independently affect the outcome of the decision [12]. 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 (9) 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

$$\forall d \in \mathrm{D}(P)\ \forall c \in \mathrm{C}(d)\ \exists td_1, td_2 \in \mathbb{TD}.\ unique\_Cause(c, d, td_1, td_2). \tag{9}$$

$$\begin{aligned} unique\_Cause(c_1, d, td_1, td_2) \Longrightarrow\ & control\_Expr(td_1, td_2, c_1) \wedge \\ & control\_Expr(td_1, td_2, d) \wedge \\ & \forall c_2 \in \mathrm{C}d.\ (c_2 \neq c_1) \longrightarrow \\ & is\_invariantExpr(\{td_1, td_2\}, c_2). \end{aligned} \tag{10}$$

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 dataset $IV_T(x)$ 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:

$$\begin{aligned} control\_Expr(td_1, td_2, x) \Longrightarrow\ & (td_1 \in IV_T(x) \wedge td_2 \in IV_F(x)) \vee \\ & (td_2 \in IV_T(x) \wedge td_1 \in IV_F(x)). \end{aligned} \tag{11}$$

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:

$$\begin{aligned} is\_invariantExpr(ID, x) \Longrightarrow\ & (ID \cap IV_T(x) = \emptyset) \vee (ID \cap IV_F(x) = \emptyset). \end{aligned} \tag{12}$$

The above definition of MCDC is the original definition given in the RTCA/DO178b document [12]. However, this definition is rather strict, so that people thought of some less restrictive definitions. For example, it is not possible with the original definition to cover a decision with strongly coupled conditions. (Two conditions c_1, c_2 are strongly coupled, iff they have the same input data partitioning for their satisfiability valuation, i.e., $(IV_T(c_1) = IV_T(c_2) \wedge IV_F(c_1) = IV_F(c_2)) \vee (IV_T(c_1) = IV_F(c_2) \wedge IV_F(c_1) = IV_T(c_2))$.) As described in [25], there exist at least three definitions of MCDC:

(i) *Unique-Cause MCDC*: this is the original definition given in [12].

(ii) *Unique-Cause + Masking MCDC*: this definition of MCDC is less restrictive as it requires in case of strongly coupled conditions to test only that one of them covers the decision (masking) [15].

(iii) *Masking MCDC*: this is less restrictive than the two above, as it does not require the *Unique-Cause*. A condition is masked if its value cannot influence the outcome of a decision due to the overruling values of other conditions. For *Masking MCDC* it is sufficient to show that each condition can affect the outcome of the decision without being masked. However, *Masking MCDC* is not required to test whether conditions do independently cover the decision. It focuses more on testing the correct implementation of subexpressions within a Boolean expression.

According to Chilenski the metric *Masking MCDC* should be the preferred form of MCDC as it provides the same error detection probability but allows for more different test sets and thus the generation of test data more is cost-effective [25].

Within this article we focus on the original definition of MCDC. 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.

Multiple Condition Coverage (MCC). Requires besides DC and CC that each possible combination of outcomes of the conditions of each decision is executed at least once. MCC demands a rather high number of test cases: to achieve full MCC of a decision with n conditions 2^n tests are necessary. MCC is desired in theory, but MCC tends to be infeasible for industrial code, because there are too many conditions per decision [27]. Thus, in this work we do not address MCC.

Path Coverage (PC). Requires that each path of a program P has been tested at least once. Since the number of paths within a program typically grows exponentially with the program size (PC is even stronger than MCC), we do not address PC.

Scoped Path Coverage (SPC). Is a coverage metric recently introduced by the authors. We use this type of code coverage for *measurement-based timing analysis* [7]. In this article we formalize this coverage metric to reason about necessary properties of a compilation profile for preserving SPC. The basic idea of SPC is to partition the program P into program scopes and cover all possible paths within each program scope. Actually, PC is just the special case of using SPC in combination with only one program scope covering the whole program P.

The appropriate partitioning of a program into program scopes depends on the concrete testing goal. For example, in case of our research on *measurement-based timing analysis* [7] we use the partitioning of the program into scopes to achieve a compromise between precision of measurement results (the larger the segments the more precise) and number of necessary measurements.

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 the path. Using our above definitions, we can formally define SPC as follows:

$$\begin{aligned} &\forall ps \in \mathrm{PS}(P)\ \forall pp \in \mathrm{PP}(ps)\ \exists td \in \mathbb{TD}. \\ &\qquad (IV_R(\mathrm{B_S}(pp)) \cap \{td\}) \neq \emptyset \wedge \\ &\qquad \forall c_T \in \mathrm{C}_T(pp).\ (IV_T(c_T) \cap \{td\}) \neq \emptyset \wedge \\ &\qquad \forall c_F \in \mathrm{C}_F(pp).\ (IV_F(c_F) \cap \{td\}) \neq \emptyset. \end{aligned} \tag{13}$$

Note, that the condition "$(IV_R(\mathrm{B_S}(pp)) \cap \{td\}) \neq \emptyset$" of (13) 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.

Whether SPC is feasible in practice, depends on the program complexity itself and also on the application-specific partitioning of a program into program scopes.

Examples of test vectors sufficient for full coverage according to the different coverage metrics are given in Table 1. The ANSI C code example is a decision including three conditions. Note that in C the operators || and && influence the control flow of the program due to the short-circuit evaluation in ANSI C. This small example is meant to support the definition of different variants of coverage metric. It is not meant to show the relative costs of the different variants of structural coverage metric.

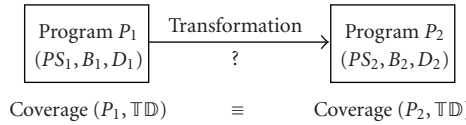

Figure 1: Coverage-preserving program transformation.

The *condition coverage* (CC) needs a relative high number of test vectors. This is because of test vectors that enforce the entering of a program decision do not necessarily enforce the execution of a specific condition within the decision. *Multiple condition coverage* (MCC) has a relative high cost for testing a single decision. However, when looking at the whole program, then *path coverage* (PC) is typically much more complex, and depending on the definition of program scopes, *scoped path coverage* (SPC) requires significantly less test vectors than PC.

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 1. Of course if a program will be transformed, also the sets of basic blocks B, the set of program decisions D, or program scopes PS may get changed. As shown in Figure 1, 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 by the test-data set $\mathbb{TD}$ is preserved to the transformed program. Based on these properties one can adjust a source-to-source transformer or a compiler to use only those optimizations that preserve the intended structural code coverage.

These coverage-preservation properties to be maintained have to ensure that whenever the code coverage is fulfilled at the original program by some test data $\mathbb{TD}$ then this coverage is also fulfilled at the transformed program with the same test data:

$$\forall \mathbb{TD}.\ coverage(P_1, \mathbb{TD}) \Longrightarrow coverage(P_2, \mathbb{TD}). \tag{14}$$

The code coverage preservation can be applied on any type of code transformation, for example, on a source-to-source transformer or a compiler.

In the first step, we have to determine for each code transformation of the code transformer whether it preserves a given structural code coverage. We call this the *coverage*

Table 1: Example: Sufficient test vectors per coverage metric.

Test vector			ANSI C expression	Coverage criterion						
A	B	C	`if(A || (B && C))`	SC	DC	CC	MCDC	MCC	PC	SPC
F	F	F	F					×		
F	F	T	F			×	×	×	×	×
F	T	F	F		×	×	×	×	×	×
F	T	T	T			×	×	×	×	×
T	F	F	T	×	×	×	×	×	×	×
T	F	T	T					×		
T	T	F	T					×		
T	T	T	T					×		

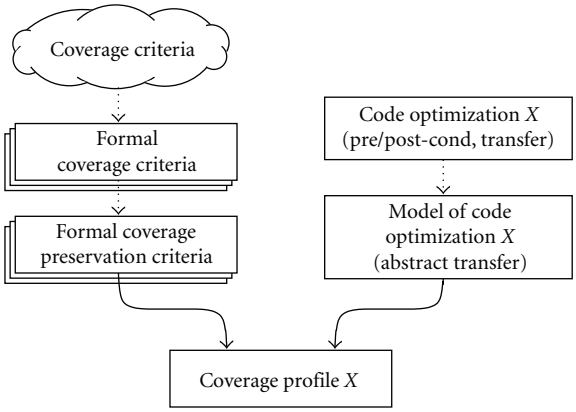

Figure 2: Determination of a coverage profile.

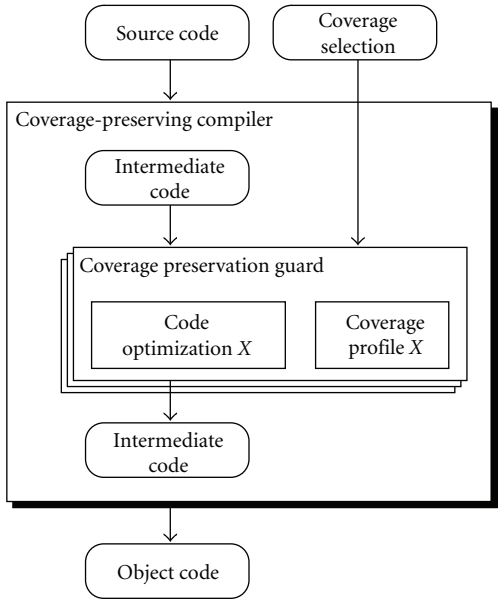

Figure 3: Application of a coverage profile.

profile of a code transformation. The determination of the coverage profile is shown in Figure 2. The structural code coverage metrics of interest have to be formalized and based on that the coverage preservation criteria have to be determined. The coverage preservation criteria together with description of a code optimization are used to calculate the coverage profile of that optimization. The construction of a formal model of the code optimization in Figure 2 is an intermediate step that is necessary if one wants to use formal verification to determine the coverage profile. In case the coverage profile is determined manually, such a formal model of the code optimization is not needed.

In the second step, the coverage preservation has to be integrated into the code transformer. As an example we assume the code transformer is a compiler, as shown in Figure 3. This coverage-preserving compiler will have an input parameter to set the code coverage metric to be preserved. The coverage-preserving compiler can have two operation modes.

Safe Mode. In this mode the coverage-preserving compiler will apply only those code optimizations that preserve the given code coverage metric. With this operation mode we assure coverage preservation at the cost of a potential degradation of performance.

Full-Optimization Mode. In this mode the coverage-preserving compiler will apply all code transformations but it will emit a warning whenever a code transformation has been used that does not ensure the preservation of the given coverage metrics. The warning message should be as specific as possible to support the user in determining additional test data to regain code coverage for the optimized code.

The determination of the coverage profile for a given code transformation and the realization of a coverage-preserving compiler are not the focus of this article. Within this article we present the foundation for such a coverage preservation framework and discuss issues that challenge its applicability.

In the following we present coverage preservation criteria for several variants of structural code-coverage metrics. The important aspect is that these preservation criteria are independent of the concrete test data $\mathbb{TD}$ that achieve the structural code coverage at the original program.

4.1. Preserving Statement Coverage (SC). Equation (15) of Theorem 4.1 provides a coverage preservation criterion for statement coverage. Equation (15) essentially says that for each basic block b' of the transformed program there exists

a basic block b of the original program such that reaching b with a given test vector implies that also b' is reached with the same test vector.

Theorem 4.1 (Preservation of SC). *Assuming that a set of test data $\mathbb{TD}$ achieves statement coverage on a given program P_1, then* (15) *provides a sufficient—and without further knowledge about the program and the test data (there is now knowledge about the test data or the program assumed), also necessary—criterion for guaranteeing preservation of statement coverage on a transformed program P_2.*

$$\forall b' \in B(P_2)\ \exists b \in B(P_1).\ IV_R(b') \supseteq IV_R(b). \tag{15}$$

Proof. Preservation of SC: Part 1, showing sufficiency: Since $\mathbb{TD}$ is assumed to achieve SC on P_1, it holds for each $b \in B_1$ that $(IV_R(b) \cap \mathbb{TD}) \neq \emptyset$. Since (15) states that $IV_R(b') \supseteq IV_R(b)$ it follows that for each $b' \in B_2$ we also have $(IV_R(b') \cap \mathbb{TD}) \neq \emptyset$. Thus, SC is preserved at P_2.

Part 2, showing necessity by indirect proof: Assuming there exists a basic block $b' \in B_2$ of P_2 such that for all basic blocks $b \in B_1$ of P_1 it holds that $\neg(IV_R(b') \supseteq IV_R(b))$, then each $IV_R(b)$ contains at least one input that is not in $IV_R(b')$. If $\mathbb{TD}$ consists of exactly those inputs, then $b\prime$ is never reached although SC holds in P_1, which implies that SC is not preserved. □

4.2. Preserving Condition Coverage (CC). To define a coverage preservation criterion for CC (Theorem 4.2) we use the auxiliary predicate $touches\_ID(x, ID)$ given in (16).

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

$$\begin{aligned} &touches\_ID(x, ID) \Longrightarrow \\ &\qquad (IV_T(x) \subseteq ID)\ \vee (IV_F(x) \subseteq ID). \end{aligned} \tag{16}$$

Equation (17) states that for each condition c' of the transformed program there exists at least one condition of the original program whose coverage implies that c' evaluates to TRUE and there exists at least one condition of the original program whose coverage implies that $c\prime$ evaluates to FALSE.

Theorem 4.2 (Preservation of CC). *Assuming that a set of test data $\mathbb{TD}$ achieves condition coverage on a given program P_1, then* (17) *provides a sufficient—and without further knowledge about the program and the test data, also necessary—criterion for guaranteeing preservation of condition coverage on a transformed program P_2:*

$$\begin{aligned} &\forall c' \in C(P_2). \\ &\qquad \exists c \in C(P_1).\ touches\_ID(c, IV_T(c')) \wedge \\ &\qquad \exists c \in C(P_1).\ touches\_ID(c, IV_F(c')). \end{aligned} \tag{17}$$

Proof. Preservation of CC: Part 1, showing sufficiency: Since $\mathbb{TD}$ is assumed to achieve CC on P_1, it holds for each $c \in \mathrm{C}(P_1)$ that $(IV_T(c) \cap \mathbb{TD}) \neq \emptyset$ and $(IV_F(c) \cap \mathbb{TD}) \neq \emptyset$. Since (17) states that for each $c' \in \mathrm{C}(P_2)$ it holds that

$$\begin{aligned} &\exists c \in \mathrm{C}(P_1).\ (IV_T(c') \supseteq IV_T(c) \vee IV_T(c') \supseteq IV_F(c)), \\ &\exists c \in \mathrm{C}(P_1).\ (IV_F(c') \supseteq IV_F(c) \vee IV_F(c') \supseteq IV_T(c)), \end{aligned} \tag{18}$$

it follows that for each $c' \in \mathrm{C}(P_2)$ we also have

$$(IV_T(c') \cap \mathbb{TD}) \neq \emptyset, \qquad (IV_F(c') \cap \mathbb{TD}) \neq \emptyset. \tag{19}$$

Thus, CC is preserved at P_2.

Part 2, showing necessity by indirect proof: Assuming there exists a condition $c' \in \mathrm{C}(P_2)$ of program P_2 such that for all conditions $c_1, c_2 \in \mathrm{C}(P_1)$ of program P_1 it either holds that

(a) $\neg(IV_T(c') \supseteq IV_T(c_1) \vee IV_T(c') \supseteq IV_F(c_1))$,

(b) $\neg(IV_F(c') \supseteq IV_F(c_2) \vee IV_F(c') \supseteq IV_T(c_2))$,

then it is possible that

(a) $\forall c \in \mathrm{C}(P_1)$: $\mathbb{TD} \cap IV_T(c') \cap (IV_T(c) \cup IV_F(c)) = \emptyset$,

(b) $\forall c \in \mathrm{C}(P_1)$: $\mathbb{TD} \cap IV_F(c') \cap (IV_F(c) \cup IV_T(c)) = \emptyset$

which in both cases violates the preservation of CC. □

Simplification of the CC Preservation Criteria. The goal of defining the coverage preservation criterion is to decide for a set of code transformations whether they could potentially disrupt the structural code coverage achieved on the original program. Typically, when checking the preservation of structural code coverage, one would simplify (17) by just checking whether each condition $c \in \mathrm{C}(P_1)$ is kept equal or simply is inverted. This would result in the simpler criterion given in (20)

$$\begin{aligned} &\forall c' \in \mathrm{C}(P_2)\ \ \exists c \in \mathrm{C}(P_1). \\ &\qquad (IV_T(c') = IV_T(c)) \vee (IV_T(c') = IV_F(c)). \end{aligned} \tag{20}$$

Working with the simple constraint of (20) may be sufficient in practice when analyzing the effect of concrete code transformations, since many transformations do not modify the conditions within a decision, but only their grouping into decisions. The simplified criterion is sufficient to allow only such code transformations that do not introduce new conditions with new unique satisfiability by the test data. Further, some transformations just invert a condition, which can be checked also with this simplified criterion.

4.3. Preserving Decision Coverage (DC). To define a coverage preservation criterion for DC (Theorem 4.3) we use the auxiliary predicate $touches\_ID(x, ID)$ given in (16), which is also used for preserving CC.

Equation (21) of Theorem 4.3 provides a coverage preservation criterion for decision coverage. Equation (21) essentially says that for each decision d' of the transformed program there exists at least one decision of the original program whose coverage implies that d' evaluates to TRUE and there exists at least one decision of the original program whose coverage implies that d' evaluates to FALSE.

Theorem 4.3 (Preservation of DC). *Assuming that a set of test data $\mathbb{TD}$ achieves decision coverage on a given program P_1, then* (21) *provides a sufficient—and without further knowledge about the program and the test data, also necessary—criterion for guaranteeing preservation of decision coverage on a transformed program P_2*

$$\forall d' \in D(P_2).$$

$$\exists d \in D(P_1).\ touches\_ID(d, IV_T(d')) \wedge \quad (21)$$

$$\exists d \in D(P_1).\ touches_{ID(d, IV_F(d'))}.$$

Proof. Preservation of DC: Part 1, showing sufficiency: since $\mathbb{TD}$ is assumed to achieve DC on P_1, it holds for each $d \in \mathrm{D}(P_1)$ that $(IV_T(d) \cap \mathbb{TD}) \neq \emptyset$ and $(IV_F(d) \cap \mathbb{TD}) \neq \emptyset$. Since (21) states that for each $d' \in \mathrm{D}(P_2)$

(1) $\exists d \in \mathrm{D}(P_1).\ (IV_T(d') \supseteq IV_T(d) \vee IV_T(d') \supseteq IV_F(d))$,

(2) $\exists d \in \mathrm{D}(P_1).\ (IV_F(d') \supseteq IV_F(d) \vee IV_F(d') \supseteq IV_T(d))$

it follows that for each $d' \in \mathrm{D}(P_2)$ we also have $(IV_T(d') \cap \mathbb{TD}) \neq \emptyset$ and $(IV_F(d') \cap \mathbb{TD}) \neq \emptyset$. Thus, DC is preserved at P_2.

Part 2, showing necessity by indirect proof: assuming there exists a decision $d' \in \mathrm{D}(P_2)$ such that for all conditions $d_1, d2 \in \mathrm{D}(P_1)$ it either holds that

(a) $\neg(IV_T(d') \supseteq IV_T(d_1) \vee IV_T(d') \supseteq IV_F(d_1))$, or

(b) $\neg(IV_F(d') \supseteq IV_F(d_2) \vee IV_F(d') \supseteq IV_T(d_2))$,

then it is possible that

(a) $\forall d_1 \in \mathrm{D}(P_1) : \mathbb{TD} \cap IV_T(d') \cap (IV_T(d_1) \cup IV_F(d_1)) = \emptyset$, or

(b) $\forall d_2 \in \mathrm{D}(P_1) : \mathbb{TD} \cap IV_F(d') \cap (IV_F(d_2) \cup IV_T(d_2)) = \emptyset$,

which in both cases violates the preservation of DC. □

Guaranteeing Decision Coverage. Guaranteeing the preservation of a structural code coverage criterion that depends on the coverage of decisions of a program is challenging, since there are many ways to re-group conditions into hierarchies of decisions without changing the program semantics.

The criterion given in (21) imposes quite strong restrictions on the performed code transformations, since it requires that for each decision $d' \in \mathrm{D}(P_2)$ there is an adequate decision $d \in \mathrm{D}(P_1)$ of the original program such that *decision coverage* is preserved. For example, consider the following code transformation:

```
if (a==3&&b==2) {
    c();
}
```

inlined style

$\Longrightarrow$

```
if (a==3) {
   if (b==2) {
      c();
   }
}
```

noninlined style.

Such a transformation is quite typical when source-code is transformed into assembly code. Actually, the only decision in the original code is (`a==3 && b==2`). Having *decision coverage* on the original code, there are numerous code transformations possible that do not preserve *decision coverage*.

Thus, it would be useful to have another criterion to guarantee decision coverage at the transformed program. Equation (22) provides a sufficient criterion for guaranteeing decision coverage on the transformed program, assuming that *condition coverage* is fulfilled on the original program

$$\forall d' \in \mathrm{D}(P_2).$$

$$\exists c \in \mathrm{C}(P_1).\ touches\_ID(c, IV_T(d')) \wedge \quad (22)$$

$$\exists c \in \mathrm{C}(P_1).\ touches\_ID(c, IV_F(d')).$$

The new criterion requires a different, but not stronger, structural code coverage at the original code to guarantee *decision coverage* at the transformed code. This criterion is typically more flexible when generating assembly code (which typically does not have control-flow statements with complex decisions). Further, in case that *condition decision coverage* (CDC) is fulfilled at the original program, one may chose between the criteria of (21) and (22) to guarantee *decision coverage* at the transformed program.

4.4. Preserving MCDC. Preserving MCDC coverage on a transformed program is especially challenging, since the code transformation may produce arbitrary groupings of conditions into decisions. Especially the requirement that each condition can independently influence the outcome of its conditions, is rather complex to check.

As the MCDC coverage preservation criterion is rather complex, we derive them in two steps. First, we describe a rather naive criterion that is relatively ease to understand. This criterion is sufficient but not necessary (too strict). Second, we describe a "realistic" (more detailed) criterion that is sufficient and necessary.

A Naive Coverage Preservation Criterion. A sufficient but not necessary coverage preservation criterion for MCDC is given in (23). The predicate symbol *unique_Cause* is used in the same way as the real criterion: it is used to express that only

input data that fulfill MCDC at the original program have to be considered for coverage preservation

$$\begin{aligned}\forall d' \in \mathrm{D}(P_2)\ \ \forall c' \in \mathrm{C}(d')\ \exists d \in \mathrm{D}(P_1)\ \ \exists c \in \mathrm{C}(d).\\ \exists \langle id_1, id_2\rangle \in \mathbb{TD}.\\ unique\_Cause(c, d, id_1, id_2) \Longrightarrow\\ unique\_Cause(c\prime, d\prime, id_1, id_2).\end{aligned} \tag{23}$$

This naive criterion is not necessary since it requires the coverage preservation of the conditions in the transformed program P_2 by a single condition from the original program P_1.

Another drawback of this naive criterion is that it is based on a concrete set of test data $\mathbb{TD}$ that are used to achieve MCDC at the original program. To ensure coverage preservation in general, it would be necessary to ensure that the criterion holds for all possible sets of test data $\mathbb{TD}$ that achieve MCDC at the original program, which tends to be intractable in practice.

A Realistic Coverage Preservation Criterion. To define an easier testable (but more complicated) coverage preservation criterion for MCDC (Theorem 4.4) we use the auxiliary predicate $mult\_control\_Expr(ID_1, ID_2, x)$ given in (24). The predicate $mult\_control\_Expr$ is similar to the predicate symbol $control\_Expr(td_1, td_2, x)$, with the difference that it performs the control check on all members of two sets of input data. The predicate $mult\_control\_Expr(ID_1, ID_2, x)$ is used for the coverage preservation of MCDC to test whether the condition x of the original program refers to TRUE for one input data set ID_1 or ID_2 and refers to FALSE for the other. Besides $mult\_control\_Expr(ID_1, ID_2, x)$, also the predicate $unique\_Cause(c_1, d, td_1, td_2)$ (10) is used to describe the preservation criterion for MCDC coverage

$$\begin{aligned}mult\_control\_Expr(ID_1, ID_2, x) \Longrightarrow\\ (ID_1 \subseteq IV_T(x) \wedge ID_2 \subseteq IV_F(x)) \vee\\ (ID_1 \subseteq IV_F(x) \wedge ID_2 \subseteq IV_T(x)).\end{aligned} \tag{24}$$

The criterion given in equ_preserve_mcdc states that for each condition c' of a decision d' of the transformed program there exist two sets of input data ID_1 and ID_2 whose members achieve the *unique_Cause* criterion needed for MCDC coverage. Further, there has to be a condition of the original program such that the ID_1 is a subset of either the true-satisfiability valuation or the false-satisfiability valuation (tested with the predicate $mult\_control\_Expr$). ID_2 the same requirement as ID_1.

Theorem 4.4 (Preservation of MCDC). *Assuming that a set of test data $\mathbb{TD}$ achieves MCDC coverage on a given program P_1, then* (25) *provides a sufficient—and without further knowledge about the program and the test data, also necessary—criterion for guaranteeing preservation of MCDC coverage on a transformed program P_2*

$$\begin{aligned}&\forall d' \in D(P_2)\ \ \forall c' \in C(d')\ \exists ID_1, ID_2 \subseteq \mathbb{ID}.\\ &\quad \Big(\exists d \in D(P_1)\ \ \exists c \in C(d)\ \exists ID_{tmp} \subseteq \mathbb{ID}.\\ &\qquad mult\_control\_Expr\Big(ID_1, ID_{tmp}, c\Big) \wedge\\ &\qquad \forall \langle id_1, id_2\rangle \in ID_1 \times ID_{tmp}.\\ &\qquad\quad unique\_Cause(c, d, id_1, id_2)\Big)\\ &\quad \wedge\\ &\quad \Big(\exists d \in D(P_1)\ \ \exists c \in C(d)\ \exists ID_{tmp} \subseteq \mathbb{ID}.\\ &\qquad mult\_control\_Expr\Big(ID_2, ID_{tmp}, c\Big) \wedge\\ &\qquad \forall \langle id_1, id_2\rangle \in ID_2 \times ID_{tmp}.\\ &\qquad\quad unique\_Cause(c, d, id_1, id_2)\Big)\\ &\quad \wedge\\ &\quad \forall \langle id_1, id_2\rangle \in ID_1 \times ID_2.\\ &\qquad unique\_Cause(c', d', id_1, id_2)).\end{aligned} \tag{25}$$

Proof. Preservation of MCDC: Part 1, showing sufficiency: Since $\mathbb{TD}$ is assumed to achieve MCDC on P_1, it holds for each $d \in \mathrm{D}(P_1)$ and for each $c \in \mathrm{C}(d)$ that there exist at least two test vectors $td_1, td_2 \in \mathbb{TD}$ such that $unique\_Cause(c, d, td_1, td_2)$. Since $unique\_Cause(c, d, td_1, td_2)$ as defined in (10) for each condition is the formal definition of MCDC it directly follows that

$$\begin{aligned}\forall d' \in \mathrm{D}(P_2)\ \ \forall c' \in \mathrm{C}(d')\ \exists ID_1, ID_2 \subseteq \mathbb{ID}.\\ \cdots \wedge \cdots \wedge\\ \forall \langle id_1, id_2\rangle \in ID_1 \times ID_2.\\ unique\_Cause(c', d', id_1, id_2))\end{aligned} \tag{26}$$

is a sufficient criterion to ensure that MCDC is preserved at program P_2.

Part 2, showing necessity by indirect proof: Assuming there exists a decision $d\prime \in \mathrm{D}P_2$ with a condition $c' \in \mathrm{C}(d')$ such that for all input-data subsets $ID_1, ID_2, \subseteq \mathbb{ID}$ it either holds that

$$\begin{aligned}&\text{(a)}\ \forall d \in \mathrm{D}(P_1)\ \forall c \in \mathrm{C}(d)\ \forall ID_{tmp} \subseteq \mathbb{ID}.\\ &\quad \neg mult\_control\_Expr\Big(ID_1, ID_{tmp}, c\Big),\\ &\text{(b)}\ \forall d \in \mathrm{D}(P_1)\ \forall c \in \mathrm{C}(d)\ \forall ID_{tmp} \subseteq \mathbb{ID}.\\ &\quad \exists \langle id_1, id_2\rangle \in ID_1 \times ID_{tmp}.\\ &\quad \neg unique\_Cause(c, d, id_1, id_2),\ \text{or}\\ &\text{(c)}\ \exists \langle id_1, id_2\rangle \in ID_1 \times ID_2.\\ &\quad \neg unique\_Cause(c', d', id_1, id_2),\end{aligned} \tag{27}$$

then it is possible that

$$\begin{aligned}&(\text{a})\ \forall d \in \mathrm{D}(P_1)\ \forall c \in \mathrm{C}(d)\ \forall TD_1, TD_2 \subseteq \mathbb{TD}.\\ &\quad \neg mult\_control\_Expr(TD_1, TD_2, c), \text{ or}\end{aligned} \tag{28}$$

(for all conditions in the original program P_1 condition coverage is not fulfilled; this case is already excluded by assumption of having MCDC coverage at P_1)

$$\begin{aligned}&(\text{b})\ \forall d \in \mathrm{D}(P_1)\ \forall c \in \mathrm{C}(d)\ \forall td_1, td_2 \in \mathbb{TD}.\\ &\quad \neg unique\_Cause(c, d, td_1, td_2),\end{aligned} \tag{29}$$

(there is no MCDC coverage at the original program P_1; this case is already excluded by assumption of having MCDC coverage at P_1)

$$\begin{aligned}&(\text{c})\ \exists d' \in \mathrm{D}(P_2)\ \exists c' \in \mathrm{C}(d')\ \forall td_1, td_2 \in \mathbb{TD}.\\ &\quad \neg unique\_Cause(c', d', td_1, td_2),\end{aligned} \tag{30}$$

(the test data $\mathbb{TD}$ do not provide MCDC coverage at the transformed program P_2) which in each case violates the preservation of MCDC: Case (a) and (b) violate the preservation of MCDC since they are in contradiction with the requirement that MCDC is achieved at the original program. Case (c) states that there exists a condition in the transformed program for which there are no test data to achieve *unique cause* coverage, which is required for MCDC. □

4.5. Preserving Scoped Path Coverage (SPC). To define a coverage preservation criterion for SPC (Theorem 4.5) we use the auxiliary predicate $is\_CondTF\_enclosed(ID, C_T, C_F)$ given in (31).

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

$$\begin{aligned}&is\_CondTF\_enclosed(ID, C_T, C_F) \Longrightarrow\\ &\qquad \exists c_T \in C_T.\ IV_T(c_T) \subseteq ID \vee\\ &\qquad \exists c_F \in C_F.\ IV_F(c_F) \subseteq ID.\end{aligned} \tag{31}$$

As stated in Theorem 4.5, (32) provides a coverage preservation criterion for SPC. Equation (32) says that for each scoped path pp' of the transformed program there exists a scoped 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 (32) states that for each condition c' of pp' that has to be evaluated to TRUE, there exists a condition c of a scoped path in the original program that will imply the True evaluation of c' (by predicate *is_CondTF_enclosed*). Finally, Equation (32) states that for each condition c' of pp' that has to be evaluated to FALSE, there exists a condition c of a scoped path in the original program that will imply the FALSE evaluation of c' (by predicate *is_CondTF_enclosed*).

Theorem 4.5 (Preservation of SPC). *Assuming that a set of test data $\mathbb{TD}$ achieves scoped path coverage on a given program P_1, then* (32) *provides a sufficient—and without further knowledge about the program and the test data, also necessary—criterion for guaranteeing preservation of scoped path coverage on a transformed program P_2*

$$\begin{aligned}&\forall ps' \in PS(P_2)\ \forall pp' \in PP(ps').\\ &(\exists ps \in PS(P_1)\ \exists pp \in PP(ps).\\ &IV_R(B_S(pp')) \supseteq IV_R(B_S(pp))) \wedge\\ &(\forall c' \in C_T(pp')\ \exists ps \in PS(P_1)\ \exists pp \in PP(ps).\\ &is\_CondTF\_enclosed(IV_T(c'), C_T(pp), C_F(pp))) \wedge\\ &(\forall c' \in C_F(pp')\ \exists ps \in PS(P_1)\ \exists pp \in PP(ps).\\ &is\_CondTF\_enclosed(IV_F(c'), C_T(pp), C_F(pp))).\end{aligned} \tag{32}$$

Proof. Preservation of SPC: Part 1, showing sufficiency: Since $\mathbb{TD}$ is assumed to achieve SPC on P_1, it holds for each $c_T \in \mathrm{C}_T(pp)$ and each $c_F \in \mathrm{C}_F(pp)$ with $pp \in \mathrm{PP}(ps) \wedge ps \in \mathrm{PS}(P_1)$ that there exists test data $td \in \mathbb{TD}$ with

$$\begin{aligned}&(IV_R(\mathrm{B}_S(pp)) \cap \{td\}) \neq \emptyset) \wedge\\ &(\forall c_T \in \mathrm{C}_T(pp).\ (IV_T(c_T) \cap \{td\}) \neq \emptyset) \wedge\\ &(\forall c_F \in \mathrm{C}_F(pp).\ (IV_F(c_F) \cap \{td\}) \neq \emptyset).\end{aligned} \tag{33}$$

Since (32) states that

$$\begin{aligned}&(\exists ps \in \mathrm{PS}(P_1)\ \exists pp \in \mathrm{PP}(ps).\\ &\qquad IV_R(\mathrm{B}_S(pp')) \supseteq IV_R(\mathrm{B}_S(pp)))\end{aligned} \tag{34}$$

it follows that

$$((IV_R(\mathrm{B}_S(pp')) \cap \{td\}) \neq \emptyset). \tag{35}$$

As (32) also states that

$$\begin{aligned}&\forall c' \in \mathrm{C}_T(pp')\ \exists ps \in \mathrm{PS}(P_1)\ \exists pp \in \mathrm{PP}(ps).\\ &is\_CondTF\_enclosed(IV_T(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp))\end{aligned} \tag{36}$$

it follows that

$$(\forall c'_T \in \mathrm{C}_T(pp').\ (IV_T(c'_T) \cap \{td\}) \neq \emptyset). \tag{37}$$

Finally, as (32) states that

$$\begin{aligned}&\forall c' \in \mathrm{C}_F(pp')\ \exists ps \in \mathrm{PS}(P_1)\ \exists pp \in \mathrm{PP}(ps).\\ &is\_CondTF\_enclosed(IV_F(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp))\end{aligned} \tag{38}$$

it follows that

$$(\forall c'_F \in \mathrm{C}_F(pp').\ (IV_F(c'_F) \cap \{td\}) \neq \emptyset). \quad (39)$$

Thus, SPC is preserved at P_2.

Part 2, showing necessity by indirect proof: Assuming there exists a scoped program path $pp' \in \mathrm{PP}(ps') \mid ps' \in \mathrm{PS}(P_2)$ such that for all scoped program paths $pp \in \mathrm{PP}(ps) \mid ps \in \mathrm{PS}(P_1)$ it either holds that

(a) $IV_R(\mathrm{B}_S(pp')) \subset IV_R(\mathrm{B}_S(pp))$, or

(b) $\exists c' \in \mathrm{C}_T(pp').$

$\neg is\_CondTF\_enclosed(IV_T(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp))$, or

(c) $\exists c' \in \mathrm{C}_F(pp').$

$\neg is\_CondTF\_enclosed(IV_F(c'), \mathrm{C}_T(pp), \mathrm{C}_F(pp)),$ (40)

then at least one of the following cases is possible:

$$\text{(a)}\ \forall td \in \mathbb{TD}.\ (IV_R(\mathrm{B}_S(pp')) \cap \{td\}) = \emptyset) \quad (41)$$

(the first basic block of a scoped program path of the transformed program P_2 is not executed with the given test data $\mathbb{TD}$)

$$\text{(b)}\ \forall td \in \mathbb{TD}.\ (\exists c'_T \in \mathrm{C}_T(pp').\ (IV_T(c'_T) \cap \{td\}) = \emptyset) \quad (42)$$

(one of the conditions of a scoped program path of the transformed program P_2 that has to be evaluated to True evaluates to False for all test vectors of $\mathbb{TD}$)

$$\text{(c)}\ \forall td \in \mathbb{TD}.\ (\exists c'_F \in \mathrm{C}_F(pp').\ (IV_F(c'_F) \cap \{td\}) = \emptyset) \quad (43)$$

(one of the conditions of a scoped program path of the transformed program P_2 that has to be evaluated to False evaluates to True for all test vectors of $\mathbb{TD}$) which in each case violates the preservation of SPC. □

5. Application of Coverage Preservation

In Section 4 we present some formal criteria that must be fulfilled to preserve a given structural code coverage criterion. In this section we discuss how to apply these coverage-preservation criteria to permit only those code transformations that preserve structural code coverage.

There are only two types of code transformations that can disrupt the preservation of structural code coverage:

(i) transformations that change the reachability of statements or conditions. For example, reordering the conditions within a decision can change the reachability of conditions and thus also statements,

(ii) transformations that add new conditional control-flow paths into the program. For example, branch optimization can introduce new conditional branches.

```
(1) while (a>5 && b>3)        (1) while (b>3 && a>5)
(2) {                         (2) {
(3)   BODY;            ⟹     (3)   BODY;
(4) }                         (4) }
```

Figure 4: Example: condition reordering.

In the following we provide examples of how to determine with help of the introduced formalism whether a code transformation disrupts structural code coverage.

We use a small sample code and show how a concrete code transformation changes the code. We use a compact scheme to address elements of the code samples. For example, s_1 denotes the statement at source line 1. If there are multiple statements at the source line, we identify them by an additional index letter, for example, s_{1a}, s_{1b}, and so forth. Similarly, d_i and c_i address the decisions and conditions at source line i. Based on this notation we use our formalism to describe by a graph the coverage relations between different program elements. $IV_X(x_1) \rightarrow IV_Y(x_2)$ denotes that the coverage of type X at element x_1 implies the coverage of type Y at element x_1. The possible coverage types are "R" (reaching), "T" (true-evaluation) and "F" (false-evaluation). A dotted line between elements from the original and the transformed code denotes that the two elements have the same coverage. A dotted arrow from an element of the original code to an element of the transformed code denotes that a coverage of the original node by the concrete test data implies also the coverage of the element in the transformed code. Structural code coverage is achieved, if all elements of the transformed code have a connection to an element of the original code. Elements of the transformed graph, for which coverage is not preserved, are marked by a surrounding box.

5.1. Transformations That Change Reachability. In this section we will demonstrate how to identify code transformations that can disrupt structural code coverage by changing the reachability of program elements. As a case study, we have chosen *condition reordering*, a common code optimization that is typically used to enable other code optimizations. As shown in Figure 4, *condition reordering* changes the order of conditions within a decision.

The result of applying our formalism to the original and the transformed program is shown in Figure 5. As a major result we see that the element $IV_F(c_{1b})$ is surrounded by a box, which means that this coverage is not preserved during optimization. This means that the second condition at line 1 of the transformed program (`a > 5`) is not guaranteed to be evaluated to FALSE.

The coverage-relation graph in Figure 5 contains coverage types for statements, conditions, and decisions. From this graph we can conclude which structural code-coverage criteria are preserved by *condition reordering*. For example, missing the coverage of $IV_F(c_{1b})$ but not of $IV_R(c_{1b})$ implies that SC is still preserved. Also DC is preserved. However, coverage criteria that are based on the coverage of conditions are not guaranteed to be preserved. For example, preservation of CC or MCDC is not guaranteed. For simplicity, we

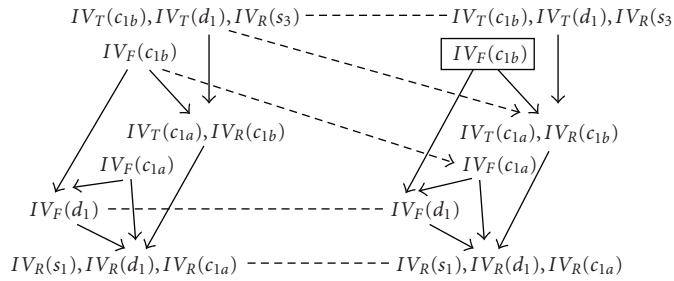

Figure 5: Coverage preservation of condition reordering.

```
(1) while (a>5 && b>3)          (1) if (a>5 && b>3)
(2) {                           (2) {
(3)   BODY;              ⟹     (3)     do {
(4) }                           (4)       BODY;
                                (5)     } while (a>5 && b>3);
                                (6) }
```

Figure 6: Example: loop inversion.

do not discuss preservation of SPC, because analyzing SPC preservation would make it necessary to unroll the program to make the different scoped program paths explicit.

5.2. Transformations That Add New Paths. In this section we will demonstrate how to identify code transformations that can disrupt structural code coverage by adding new control-flow paths to the program. As a case study, we have chosen *loop inversion*, a common code optimization that transforms a while-loop into a do-while-loop, as shown in Figure 6.

Loop inversion creates new control-flow paths. As shown in Figure 7, coverage preservation cannot be established for the exit test of the transformed loop: for the decision and conditions of source line 5 the TRUE/FALSE evaluation is not preserved. Again, this has no impact on the preservation of statement coverage. But coverage preservation of CC, DC, or MCDC is not guaranteed.

5.3. Transformations That Preserve Coverage. In this section we will demonstrate how to identify code transformations that preserve structural code-coverage. As a case study, we have chosen *loop reversal*, a typical code optimization enable further loop optimizations or vectorization of computations. As shown in Figure 8, *loop reversal* changes the condition in a program, but as we will see, it still preserves structural code-coverage.

Concluding from Figure 9, coverage preservation is achieved for all conditions, decisions, and basic blocks of the program. Thus, coverage of SC, CC, DC, and MCDC is preserved by *loop reversal*.

As a general pattern, if for each relevant expression of the transformed program there is a line or arrow to that expression originating from any expression of the same type in the original program, then the code coverage is preserved.

5.4. Modeling Hidden Control Flow. Some programming languages include statements that provide hidden control flow. For example, the logical operators && and || of ANSI C/C++ have a short-circuit evaluation semantics, which in fact already is conditional control flow. The short-circuit evaluation of ANSI C/C++ states that the second argument of the operators && and || is not evaluated if the final result of the operator is already determined by the first argument.

For example, lets look at the following code:

```
(1) if (a && (b || c)) {···}
```

Above code might be translated into the following assembly code, which demonstrates how the short-circuit evaluation of ANSI C/C++ works:

```
(1) if (!a) goto skip;
(2) if ( b) goto process;
(3) if (!c) goto skip;
(4) process:
(5)   ···
(6) skip:
```

If we do not explicitly address the hidden control-flow with the coverage preservation framework, we risk to loose full structural code-coverage as soon as the program is transformed into a new program with explicit control flow instead of the hidden, implicit control-flow.

5.5. Towards Automatic Calculation of Coverage Profiles. In this section we have demonstrated that the formalism provided in this article is helpful for determining the coverage profile of a code transformation. However, in Section 4 we argue that the calculation of the coverage profile could be done automatically, given the formal coverage preservation criterion and a formal description of the code transformation.

The code transformations can be formalized in an axiomatic semantics [28, 29], that is, by providing preconditions, postconditions, and invariants of the code transformation. Even model checking can be used to analyze code transformations [30]. There exist specific frameworks to model code transformations, for example OPTIMIX [31]. Such frameworks seem to be well-suited as a starting point for future research on automating the calculation the coverage profile of a code transformation.

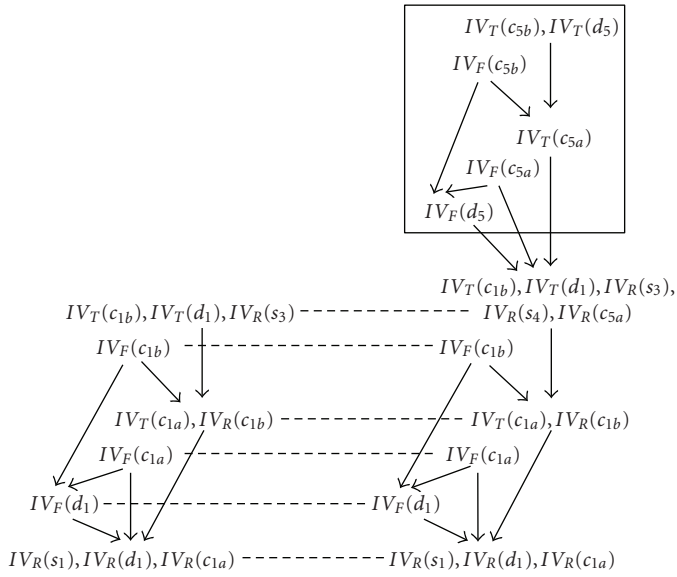

Figure 7: Coverage preservation of loop inversion.

```
(1) i=0;                          (1) i=max-1;
(2) while (i<max)                 (2) while (i≥0)
(3) {                      ⟹     (3) {
(4)    a[i]=b[i++];               (4)    a[i]=b[i--];
(5) }                             (5) }
```

Figure 8: Example: loop reversal.

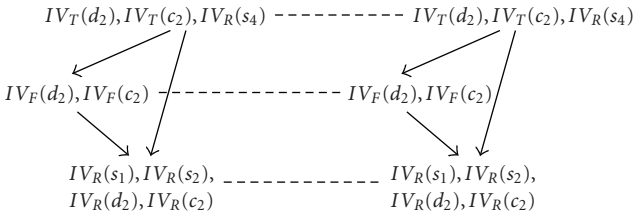

Figure 9: Coverage Preservation of Loop Reversal.

6. Challenges for Preservation of Model Coverage

The structural code-coverage criteria presented in Section 3.1 are typically applied to source code or machine code. They have been developed already before model-based development became an issue. However, these structural code-coverage criteria are also getting started to be applied to implementation models of modeling environments like *MATLAB/Simulink/Stateflow* [3, 4] from Mathworks, *Statemate* [5] from Telelogic, or *Scade* [6] from Esterel Technologies. An implementation model in general is a program implementation in a domain-specific modeling language for which automatic code generation is used to generate the source-level implementation. In this section we discuss some important differences for identifying the structural code-coverage at source-code level and implementation-model level:

(i) The modeling language may use a different implementation style (e.g., data flow instead of control flow).

(ii) The modeling language may use components of high abstraction (hiding details of complex implementation), which complicate the identification of a structural code-coverage metric's scopes within the model.

(iii) Code generation may be parametrizable (i.e., model semantics and implementation depends on the code generation settings).

(iv) Many modeling environments are under continuous development. Thus, the semantics of language constructs may change over time. Further, modeling languages are rarely standardized, often each tool provider has its own modeling language.

Though it brings in further challenges, structural code-coverage is also used for such modeling languages [19, 20]. One can still identify the control-decisions in such models, though above arguments illustrate why this can be tricky.

With the use of a structural code-coverage metric on modeling languages one can also directly use the coverage preservation criteria we presented in Section 4. To demonstrate how to do this, we examine some elements of the MATLAB/Simulink modeling environment. The main conclusions from the given examples may also apply to other modeling languages. We also discuss potential problems that lead to further research in that area.

The identified potential problems do not only apply to implementation models with automatic code generation but also to manual coding.

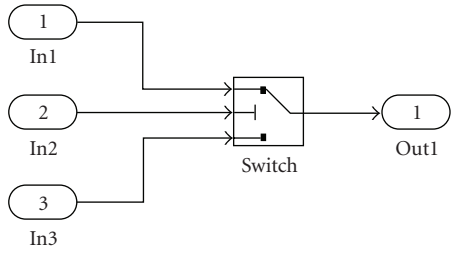

Figure 10: Simulink model switch.

```
(1) if (In2 ≥ Switch_Threshold) {
(2)    Switch = In1;
(3) } else {
(4)    Switch = In3;
(5) }
(6) Out1 = Switch;
```

Figure 11: Implementation of switch element.

6.1. A Simple Example. The *Switch* element of MATLAB/Simulink is used to control the data flow. A simple model using that element is given in Figure 10.

The semantics of the *Switch* element is best described by the code given in Figure 11 that is generated out of the simple model.

The code shows a single decision with one condition inside. In this concrete example it is valid to define the scope of decision coverage (or MCDC) as the single MATLAB/Simulink element. The generated source code for this element contains the whole decision, thus generating test data for decision coverage (or MCDC) based on the hidden control-flow of the *Switch* element results in a full decision coverage (or MCDC) at source code level.

Coverage criteria that are not based on the decisions, are easier to preserve, since coverage of conditions or basic blocks can be directly derived from the coverage of a single Matlab/Simulink element. Thus, structural code coverage like SC, CC, or SPC can be directly preserved based on the hidden control-flow.

In the following we show that it is not always that easy to apply decision-based structural code-coverage criteria like DC or MCDC to the Matlab/Simulink language and to ensure their preservation.

6.2. Identifying Code Structures in Models. In Section 6.1 we have seen how to use the implicit control flow of a data-flow element to apply a structural code-coverage metric. Given that a code generator creates explicit control flow, for example, as C code, there is the question whether it is valid to analyze Matlab/Simulink elements in isolation because the code generator can combine the logic of several data-flow elements into a single C statement. Looking only at the Matlab/Simulink elements without considering that the concrete behavior of the code generator, does not allow, for example, to identify the set of conditions that will be grouped into a single decision.

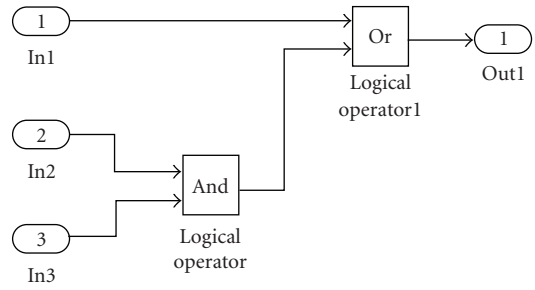

Figure 12: Simulink model expr_or_and.

To give an example, we use two logical operators as shown in Figure 12. Basically, the *logical or* as well as the *logical and* form a condition. But the choice of how to map these conditions into decisions in the generated code is left to the code generator. For example, in the non-inlined implementation variant of this model we get two decisions:

```
expr = (In2 && In3);   /* decision 1 */
Out1 = (In2 && In3);   /* decision 2 */
```

The decisions in this code example are the assignments of a logical expression to a program variable. This is just a compact form instead of using an `if/then` construct. Now lets look at the inlined implementation variant of this model, which is generated by default by the code generator *Real-Time Workshop* of Matlab 6.5.1 and Simulink 5.1. In the inlined implementation variant we get a single decision:

```
Out1 = (In1 || (In2 && In3));
```

Actually, achieving structural code coverage on the non-inlined implementation variant requires in general a smaller number of test data. For example, Rajan et al. have shown that full MCDC coverage at the non-inlined implementation variant yielded only about 13.6% MCDC coverage at the inlined implementation variant [20].

Without the knowledge about the behavior of the code generator it is not possible to formulate a preservation criterion for *decision coverage* (DC) or *MCDC coverage* that is both, sufficient and necessary independently of the exploited implementation variant. Maybe more surprisingly, this problem also arises for condition coverage (CC) because given that c_2 = (`In2 && In3`), the chosen implementation variant in this concrete example influences the set $IV_T(c_2)$ (where the condition c_2 is reached with an evaluation to True). The problem also arises for statement coverage (SC) because inlining reduces the reachability valuation of (`In2 && In3`).

6.3. The Complexity of Model Coverage. In Section 6.2 we raised the question of how much model elements have to be considered together to identify the scopes of structural code-coverage criteria. Now we argue that this scoping problem can also arise within a single element due to the rather complex code that may be generated for a Matlab/Simulink element.

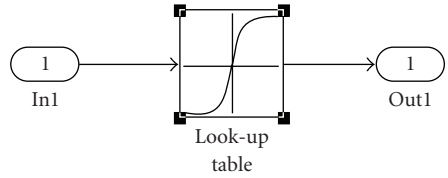

Figure 13: Simulink model lookup table.

```
(1) real Look_Up_Table;
(2) unsigned int iLeft;
(3) BINARYSEARCH_real_T_Near_iL(&iLeft, In1,
(4)         Look_Up_Table_XData, 10);
(5) Look_Up_Table = Look_Up_Table_YData[iLeft];
(6) Out1 = Look_Up_Table;
```

Figure 14: Implementation of the lookup table using binary search.

For example, consider the lookup table used in the model given Figure 13. The lookup table in this example approximates a function by using a vector for some concrete data of the x-axis of a function (`Look_Up_Table_XData`) and a vector for the corresponding data of the y-axis of the function (`Look_Up_Table_YData`). There are different ways of how to map with help of the look-up table an arbitrary input value of the approximated function to the corresponding function value. With Matlab 6.5.1 and Simulink 5.1 the code generator *Real-Time Workshop* offers the following lookup policies: *interpolation-extrapolation*, *interpolation-use end values*, *use input nearest*, *use input below*, and *use input above*. Using the lookup policy *use input nearest* we get the implementation shown in Figure 14 of the lookup table using binary search. This implementation uses the library function `BINARYSEARCH_real_T_Near_iL` given in Figure 15.

Without plotting the implementations for the other look-up policies, one can see that the *Look-Up Table* element results in rather complex control flow.

To conclude, specifying structural code-coverage criteria at the model-level that are both, sufficient and necessary, is not possible without knowledge about the behavior of the code generator.

7. Summary and Conclusion

Structural code-coverage criteria are a useful supplementary criterion to monitor the progress of testing. There are several application scenarios where the test data have been obtained at a different program representation as where the test data are executed: when using model-based testing, when the software has evolved over time, or the test-data generation is done on higher program representations to achieve easy retargetability of the test-data generation. In all such cases there is the question of whether the structural code coverage achieved at the original program representation is also fulfilled at the transformed program representation.

Within this article we analyzed the problem of preserving structural code coverage when transforming the program

```
(1) void BINARYSEARCH_real_T_Near_iL(
(2) unsigned int *piLeft, real u,
(3) const real *pData, unsigned int iHi)
(4) {
(5)   unsigned int iRght;
(6)   *piLeft = 0;
(7)   iRght = iHi;
(8)   real diffLeft, diffRght;
(9)   if (u ≤ pData[0]) {
(10)    iRght = 0;
(11)  } else if (u ≥ pData[iHi]) {
(12)    *piLeft = iHi;
(13)  } else {
(14)    unsigned int i;
(15)    while ((iRght - *piLeft) > 1)
(16)      i = (*piLeft + iRght) >> 1;
(17)      if (u < pData[i])
(18)        iRght = i;
(19)      else
(20)        *piLeft = i;
(21)    }  /* while */
(22)    diffLeft = u - pData[*piLeft];
(23)    diffRght = pData[iRght] - u;
(24)    if (diffRght ≤ diffLeft) {
(25)      *piLeft = iRght;
(26)    }  /* if */
(27)  }  /* if */
(28) }
```

Figure 15: BINARYSEARCH_real_T_Near_iL.

model or program code. We introduced a notation for formalizing structural code-coverage. This notations is convenient to also express test-data independent criteria for preserving the code coverage. These criteria may be used to prove whether a given program transformation does always preserve the structural code coverage of interest. Further, the presented preservation criteria are naturally applied on source-to-source program transformations or on optimizing program compilation. And they may be also applied on model coverage, for which we identified some additional challenges.

Future work is to develop automatic proofs to decide whether a program transformation preserves structural code coverage. In our current analysis we focused on preservation of full coverage. It is also interesting to look on preservation of partial structural code coverage.

Acknowledgments

We would like to thank Susanne Kandl and the anonymous reviewers for valuable comments on earlier versions of this article. The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Sustaining Entire Code-Coverage on Code Optimization" (SECCO) under contract P20944-N13.

References

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems: Advanced Lectures*, vol. 3472 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2005.

[2] F. Cruz, R. Barreto, L. Cordeiro, and P. Maciel, "ezRealtime: a domain-specific modeling tool for embedded hard real-time software synthesis," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 1510–1515, Munich, Germany, March 2008.

[3] The MathWorks Inc., *Using MATLAB Version 6*, The Mathworks, Natick, Mass, USA, 2002.

[4] The MathWorks Inc., *Using Simulink Version 5*, The Mathworks, Natick, Mass, USA, 2002.

[5] D. Harel, H. Lachover, A. Naamad, et al., "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990.

[6] F.-X. Dormoy, "Scade 6: a model based solution for safety critical software development," in *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS '08)*, pp. 1–9, Toulouse, France, January-February 2008.

[7] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 1–15, Porto Sani, Greece, October 2008.

[8] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-based worst-case execution time analysis using automatic test-data generation," in *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis*, pp. 67–70, Catania, Italy, June 2004.

[9] R. Kirner, "SCCP/x: a compilation profile to support testing and verification of optimized code," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '07)*, pp. 38–42, Salzburg, Austria, September-October 2007.

[10] S. S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, San Fransisco, Calif, USA, 1997.

[11] M. Whalen, A. Rajan, M. Heimdahl, and S. Miller, "Coverage metrics for requirements-based testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '06)*, pp. 25–36, Portland, Me, USA, July 2006.

[12] "Software considerations in airborne systems and equipment certification," RTCA/DO-178B, 1992.

[13] S. A. Vilkomir and J. P. Bowen, "From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria," *Formal Aspects of Computing*, vol. 18, no. 1, pp. 42–62, 2006.

[14] S. A. Vilkomir and J. P. Bowen, "Formalization of software testing criteria using the Z notation," in *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC '01)*, pp. 351–356, Honolulu, Hawaii, USA, October 2001.

[15] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[16] Object Management Group, "MDA Guide Version 1.0.1," document no. omg/2003-06-01, June 2003.

[17] M. P. E. Heimdahl, M. Whalen, A. Rajan, and S. P. Miller, "Testing strategies for model-based development," Tech. Rep. NASA/CR-2006-214307, National Aeronautics and Space Administration, Hampton, Va, USA, April 2006.

[18] A. Rajan, M. Whalen, and M. Heimdahl, "Model validation using automatically generated requirements-based tests," in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE '07)*, pp. 95–104, Dallas, Tex, USA, November 2007.

[19] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener, "The interplay between model coverage and code coverage," in *Proceedings of the 11th European International Conference on Software Testing, Analysis and Review (EuroSTAR '03)*, Amsterdam, The Netherlands, December 2003.

[20] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170, Leipzig, Germany, May 2008.

[21] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*, pp. 170–179, Florence, Italy, November 2001.

[22] M. Harman, L. Hu, R. Hierons, et al., "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[23] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass, USA, 1997.

[24] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner, "Automatic timing model generation by CFG partitioning and model checking," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 606–611, IEEE, Munich, Germany, March 2005.

[25] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Boeing Commercial Airplane Group, Seattle, Wash, USA, April 2001.

[26] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, NY, USA, 1979.

[27] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," Tech. Rep. NASA/TM-2001-210876, National Aeronautics and Space Administration, Hampton, Va, USA, May 2001.

[28] R. Floyd, "Assigning meaning to programs," in *Mathematical Aspects of Computer Science*, vol. 19 of *Proceedings of Symposia in Applied Mathematics*, pp. 19–32, American Mathematical Society, Providence, RI, USA, 1976.

[29] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[30] D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen, "Compiler optimization correctness by temporal logic," *Higher Order and Symbolic Computation*, vol. 17, no. 3, pp. 173–206, 2003.

[31] U. Aßmann, "How to uniformly specify program analysis and transformation with graph rewrite systems," in *Proceedings of the 6th International Conference on Compiler Construction (CC '96)*, P. Fritzson, Ed., pp. 121–135, Springer, Linköping, Sweden, April 1996.