FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatic Coverage-Profile Calculation for Code Optimization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Technische Informatik

eingereicht von

Walter Haas
Matrikelnummer 8125364

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Dipl.-Ing. Dr.techn. Raimund Kirner

Wien, 30.11.2010

(Unterschrift Verfasser/in)

(Unterschrift Betreuer)

Walter Haas, Richard-Genee-Strasse 4, 2500 Baden bei Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30.11.2010 _____

Abstract

For software testing or measurement based timing analysis it is often required to achieve a particular structural code-coverage criteria. It is beneficial to derive the test-data at a higher-level representation of the tested program, like source code for instance. During generation of machine code modern compilers apply optimizations to make best use of the target computer platform, for example, parallel execution or memory hierarchies. But these optimizations can destroy the structural code-coverage being achieved in the higher-level representation of the program.

A possibility to face this problem is to investigate in advance whether optimizing code transformations preserve a structural code-coverage of interest or not, and to summarize this information in a *coverage profile* for each transformation. A collection of such profiles can then be used to adjust a compiler to apply only those optimizations that preserve the intended structural code coverage or to emit warnings, whenever a code transformation does not ensure the preservation of a given structural code coverage.

This thesis develops a system for automatic analysis of code-transformations with respect to preservation of structural code coverage. The analysis is based on an existing formal coverage preservation theory. The first part of this work establishes a formal description of code-transformations. In the second part, the formalism is transposed to a mathematical software system for automatic analysis of code transformations with respect to their ability for preserving certain kinds of code coverage, and to generate a *coverage profile*. Finally, a certain number of code optimizations are provided in the third part to demonstrate the feasibility of the approach.

Keywords. Coverage metric, structural code coverage, coverage preservation, code optimization, program transformation, automatic analysis.

Zusammenfassung

Beim Testen von Software und für die Measurement Based Timing Analysis wird oft das Einhalten bestimmter Code Coverage Kriterien verlangt. Vorzugsweise werden die Testdaten aus einer höheren Repräsentationsform des getesteten Programms abgeleitet, beispielsweise aus dem Source Code. Moderne Compiler nehmen beim Erzeugen des Maschinencodes jedoch zahlreiche Optimierungen vor, um die Eigenschaften der verwendeten Zielplattform, beispielsweise durch Parallelverarbeitung oder hierarchisch organisierte Zwischenspeicher (Cache), so gut wie möglich auszunutzen. Solche Optimierungen können jedoch die strukturelle Code Coverage zerstören, die in der höheren Repräsentationsform bereits erzielt wurde.

Eine Möglichkeit dieses Problem zu umgehen ist, Code Optimierungen vorab zu analysieren, ob sie eine bestimmte Art der strukturellen Code Coverage bewahren oder nicht. So kann für jede Code Optimierung ein *Coverage Profil* erstellt werden. Ein Compiler kann die vorhandenen Profile dann benutzen, um entweder beim Übersetzungsvorgang nur Optimierungen vorzunehmen, welche die entsprechende Code Coverage erhalten, oder um eine Warnung auszugeben falls eine angewendete Optimierung die Erhaltung einer bestimmten Code Coverage nicht sicherstellt.

Diese Diplomarbeit entwickelt ein System zur automatischen Analyse von Code Transformationen in Hinblick auf die Erhaltung struktureller Code Coverage Kriterien. Die Analyse basiert auf einer bereits vorhandenen, formalen Theorie der Code Coverage Erhaltung. Der erste Teil der Arbeit führt eine formale Beschreibung von Code-Transformationen ein. Der zweite Teil überträgt den zuvor eingeführten Formalismus in ein mathematisches Software-System, um eine automatische Analyse von Code Transformationen bezüglich der Erhaltung bestimmter Arten von Code Coverage durchzuführen und daraus ein *Coverage Profil* zu erstellen. Die Anwendbarkeit des Lösungsansatzes wird schließlich im dritten Teil anhand mehrerer Analyse Beispiele gezeigt.

Schlagworte. Coverage Metrik, Strukturelle Code Coverage, Coverage Erhaltung, Codeoptimierung, Programmtransformation, Automatische Analyse.

Contents

CHAPTER 1

Introduction

1.1 Background

There is a growing number of computer-controlled systems in safety-critical real-time applications. The correctness of the functional behaviour of such systems not only depends on the correct result of a calculation. Compliance with temporal requirements is also an essential part of the functional correctness. Therefore the verification of a real-time system has to focus on the outcome of operations as well as on their temporal constraints and temporal predictability. In addition, available computer hardware becomes more and more powerful and allows integration of more complex functionality. This makes analysis of the timing behaviour for these types of systems a time consuming and error-prone task. Thus, for timing analysis measurement-based approaches are often preferred to formal methods of timing analysis [54].

The need to reduce development time and cost has also resulted in a shift towards model-based development [16]. Model based development uses executable models designed with popular tools like SIMULINK or SCADE through the development process [25]. These models form a blueprint for the automatic or manual coding of the software. The existence of such executable models can help to detect errors at early phases and eliminate them with low costs. In addition models can serve as supplementary source of information for testing [8].

For software testing as well as for measurement-based timing analysis systematic generation of suitable input-data can be a challenging problem. In an ideal world one would like to perform tests or measurements using a set containing every possible input-data combination. But creating tests for all possible input-data combinations is impractical, due to the large number of test cases needed even for small programs. For instance, consider a multiplication operation of two 8-bit integers. Each operand can hold a value from a set of 256 different values. A complete test of the operation would therefore require a total number of $256 \times 256 = 65536$ test cases to cover all possible combinations of the input-data. Thus, more realistic programs will reach very fast an astronomical number of test cases if exhaustive testing is required [41].

For restricting the set of test-data to a manageable set, some strategy is needed to avoid the generation of ineffective test-data. Such strategies fall into one of two categories: func-

tional ("black box") techniques based on the requirements written down in a program specification [55], and structural ("white box") techniques based on the computational structure of the program [52]. Structural testing strategies are generally based either on a program's control structure or on its data definitions and references [41, 52].

Test adequacy metrics are used to assess the extent to which a given set of test-data satisfies its objectives [25, 24]. Test adequacy metrics to analyze and quantify the control-flow coverage that is achieved for a given set of test-data are called *structural code coverage criteria* [31]. They can be utilized in two different ways: First as an adequacy criterion to decide whether a given test-data set is complete with respect to that criterion, and second as an explicit specification for test-data selection. Structural metrics based on control flow are most suitable for automated processing which is an important fact to deal with realistic programs containing thousands of code-lines [8].

1.2 Problem Statement

A high-level representation like source code or a formal model of the software is preferred when analyzing systems with respect to a certain coverage metric, or when using a coverage metric for creation of appropriate test data. There are several reasons for this preference. High-level representations are easier for a human to understand and their complexity is reduced compared with machine code. Source code and formal models also contain additional information about the program behaviour like directives, assertions or other meta-information. This information might get lost during generation of machine code. Another aspect is the portability of such a higher-level representation to different platforms and also the portability of the tools [31].

Using a test-data generation strategy, which is oriented on code-coverage analysis and based on a model or on source-code, creates the need to ensure that the particular coverage achieved on a higher-level representation can be transposed to machine level code in an adequate way. The higher-level representation, however, is rarely intended to accommodate details of the underlying machine architecture. A naive translation may introduce inefficiencies into the generated machine level code. Therefore code generators and compilers perform many transformations to optimize the code. They are responsible for automatically suiting code to take advantage of features like memory hierarchies, pipelining, parallel execution units and many more. The goal of such optimizations is to eliminate inefficiencies and to generate a code that is well tuned for the given architecture. To achieve this goal 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 [5].

On the contrary, structural code coverage achieved on the higher-level representation can only be guaranteed by analyzing the produced machine code with respect to that coverage criterion. One possibility to avoid this object-code analysis is to configure the compiler to omit all optimizations. But this would introduce inefficiencies into the machine level code. The responsibility to take care of best use of the computer hardware would be loaded onto the programmer with the consequence of introducing more complexity into the higher-level representation. Apart from the fact that the independence of the program from the hardware architecture will get lost,

the programmer will not always be able to perform this task since the hardware architecture is not visible at source-code level.

A more subtle solution proposed by Kirner [30] evaluates code optimizations in advance for their possible influence on structural code coverage. The essence of this analysis can be collected in a *coverage preservation profile*. A collection of such coverage preservation profiles for all kinds of possible optimizations can then serve as additional configuration input for a code generator or a compiler. The information can be profitably used in one of two ways:

1. Avoiding a certain kind of optimizing transformation if the optimization does not preserve the intended structural code coverage.
2. Display a warning if an optimizing transformation applied does not guaranty the preservation of the intended structural code coverage.

Obtaining coverage-preservation profiles in an efficient manner is an important step to turn this idea into reality. If an axiomatic formal description of the optimizing code transformation with pre-conditions, post-conditions and the invariant properties of the transformation is available, formal coverage-preservation formalisms can be used to work out coverage-preservation profiles automatically [31].

1.3 Contributions

This thesis presents a possible approach for automatically analyzing code transformations with respect to structural code coverage.

- It introduces a mathematical formalism to describe code transformations. The formalism is based on control-flow-graph models describing the program structure before and after an optimizing program transformation.
- Supplementary information associated with the nodes and edges of the control-flow-graph model support to track the changes of control flow when a program is transformed.
- Based on the formal model an analysis procedure is developed to check the ability of a code transformation to preserve certain kinds of structural code-coverage. The analysis adapts the coverage-preservation formalism described by Kirner [31] to facilitate its application with the control-flow-graph model. As a preparation step, the presented analysis method describes how to collect basic information about pre-conditions, post-conditions and properties of the investigated program transformation.
- The analysis method serves as a basis for creating a framework with the mathematical software package *Mathematica*. This framework is able to automatically collect most of the basic information needed for analysis, and it calculates coverage-preservation profiles for optimizing code-transformations defined with the presented control-flow-graph model.

In addition, the thesis presents several use cases with different kinds of optimizing code-transformations, taken from the literature about compiler design. The investigations performed in the use-cases show the feasibility of the described analysis approach.

1.4 Related Work

Code coverage criteria where originally introduced as a measure for the efficiency of software tests [41]. Application of particular structural coverage metrics is especially required for testing safety-critical software [10, 12]. Many publications deal with considerations about testing strategies [52, 42], and about the properties of different kinds of coverage metrics [15, 8, 55, 59]. Considerations about the faultfinding ability of certain coverage metrics and about the influence of program structure with respect to MCDC [25, 20, 9, 42] are also of some concern. Another area of work is to investigate how to find appropriate test data based on coverage metrics to achieve certain objectives for testing [45, 21, 54].

Several approaches have been made to establish formal representations of programs for program analysis and correctness proofs [19, 27, 44]. In addition, some formal approaches have been made to reason about programs with a view at compiler optimizations [33, 36], and validation of the produced object code [61, 32]. In relation to that, some publications deal with formalized descriptions of program transformations [51, 34], which are mainly based on graph rewriting of control flow graphs [7].

Unfortunately most of the formalisms used to describe programs and compiler optimizations are centred on the computational results of program statements. From this point of view, a transformation is correct, as long as it computes the same result for the same input. Aspects of changing the flow of control during program transformation are only rarely mentioned [61].

Investigations on preservation of structural code coverage seem to be a new field of research, started recently. Considerations done in the past are mainly focused on examining advantages or disadvantages of applying metrics on source level or object-code level [46]. Formal code coverage criteria where established in [50], using the Z notation. Conclusions especially about structural code-coverage in relation to changes in the software structure are made in [18].

The basis for the work done in this thesis is the paper of Raimund Kirner [31], which presents a collection of formal criteria checking whether a certain transformation of program code preserves a particular kind of structural code coverage or not. In addition, Kirner describes in [30] the idea of integrating compilation profiles into the compilation process to ensure that structural code coverage criteria are preserved during object code generation.

1.5 Overview of this Thesis

Chapter 2 first establishes the basic terminology. In addition, it surveys the formal definition of structural code-coverage metrics and the formal description of structural code-coverage-preservation conditions.

The first part of Chapter 3 introduces a formal mathematical model for describing code transformations based on a control-flow-graph model. It continues developing a formal procedure for using the code-coverage-preservation criteria described in Chapter 2 to judge the ability of a transformation to preserve a certain kind of structural code coverage.

Chapter 4 describes the implementation of the framework based on the mathematical computer-software system *Mathematica* for automatic application of the formalisms established in Chapter 3. In addition, this chapter describes some in-depth details on the practical implemen-

tation of the framework.

Chapter 5 presents several use cases of optimizing code transformations. The chapter demonstrates how an optimizing code transformation must be prepared for the implemented framework to enable the automatic processing of the steps needed for analysis. In addition, it describes preconditions, post-conditions and properties of the investigated transformations, and it describes how automatic coverage-preservation analysis is done for them. A summary with detailed analysis results finishes each use-case section.

Finally Chapter 6 summarizes the results and compares it with the results established in a preceding work. Some conclusions about this work and ideas for future work are finishing the chapter.

The appendix is organized in two parts. First, Appendix A provides a brief overview of the *Mathematica* programming constructs used for the implementation of the analysis framework. It is intended as help for readers not familiar with *Mathematica* to understand the programming code. Second, the most important parts of the programming code are presented in Appendix B together with a short introduction how to start execution of the implemented use cases.

1.6 Summary of Chapter 1

Some background information about structural code coverage is presented. In addition, the objectives of the thesis are described. The chapter also includes a survey of work related to structural code coverage.

CHAPTER 2

Structural Code-Coverage and Coverage Preservation

This chapter gives a brief survey on structural code-coverage criteria. It presents informal definitions of certain code-coverage criteria. For some criteria a formal definition and a formal preservation theorem are stated allowing to check, whether a program transformation preserves this code-coverage or not.

Section 2.1 starts introducing the basic terminology and the foundation of formal definitions and preservation theorems. Basic reflections about program transformations complete this section. Beginning with Section 2.2, certain structural code-coverage criteria and conditions for their preservation are explained.

2.1 Basic Terms and Definitions

Program Structure

Structural code-coverage quantifies the ability of a set of test-data to cover the execution logic of a program P. A set of test-data *achieves a certain kind of structural code-coverage* in a program, if certain artifacts of that program like, for example, basic blocks or conditions are executed in a predefined way.

A program P can be seen as a composition of program instructions like simple statements and conditions [31] specified by some formal model that reflects the control flow structure of P. Most common formalisms for such a model are *control-flow graphs* or *transition systems* [33]. Chapter 3 of this thesis introduces a control-flow-graph model tailored for analysis of structural code-coverage preservation.

A program instruction with a single point of entry and exit is called a *simple statement*. The main property of a simple statement is, that it includes no control-flow decision. Therefore a simple statement always has exactly one successor, but it may have more than one predeces-

sor. Common examples for simple statements are assignments, calculations and unconditional branches.

A program instruction with at least two points of exit is called a *control-flow statement*. Through which point of exit control-flow will continue is determined by a logical decision inside the control-flow statement. A common example for a control-flow statement is a conditional branch represented by the `if`-statement. Although control-flow statements in principle can have an arbitrary number of exits (like the `switch`-statement in C/C++, for example), this thesis restricts control-flow statements to the most common case of conditional branches with two possible exits.

A *program scope* of a program P is a connected fragment of P with well-defined interfaces for entry and exit. The set of all scopes of P is denoted as $PS(P)$. The partitioning of a program into scopes is application specific and may also include the creation of partly or fully overlapping segments [60].

A sequence of consecutive statements is called a *basic block*, if it has a single point of entry at the beginning, and when entered all statements are executed in sequence without the possibility of branch except at the end of the sequence [1]. For a given program P the symbol $B(P)$ denotes the set of all basic blocks of that program. Given a program scope $ps \in PS(P)$, then $B(ps)$ denotes the set of basic blocks of scope ps.

A *condition* is a logical expression that cannot be broken down into simpler logical expressions. The set of all conditions of a program P is denoted with $C(P)$. Typical examples for conditions available in many programming languages are relational operations, for example, like $a \neq 0$.

A *decision* is a condition or a combination of multiple conditions linked together with Boolean operators. The set of all decisions of a program P is denoted with $D(P)$. If one condition occurs more then once in a decision, each occurrence is treated like a distinct condition.

If varying the outcome of one condition of a decision also changes the outcome of another condition inside the same decision, these conditions are said to be *coupled conditions*. They are *strongly coupled* if varying the outcome of one decision always varies the outcome of the other decision for all possible input-data. They are said to be *weak coupled*, if for all possible input-data varying the outcome of one decision sometimes but not always varies the outcome of the other decision [12].

A *scoped path* pp of a program scope $ps \in PS(P)$ is a sequence of basic blocks beginning at an entry point of that scope and ending at an exit point. If the scope comprises the whole program P then, for simplicity, the term *path* is used. The symbol $B(pp)$ denotes all basic blocks along a scoped path pp and $B_S(pp)$ denotes the basic block, pp is starting with. Since loops can be part of a scope, a scoped path may include multiple instances of certain basic blocks from different iterations of the loop.

To avoid some unpleasant properties when building up a consistent theory of structural code-coverage, it is often helpful to introduce the following restrictions for the investigated programs [41]:

- The considered programs and program scopes have single points of entry and exit. The reason for this restriction is, that if a program contains multiple points for entry and exit,

the required coverage may depend on the entry-point or exit-point actually used.

- Any statement of the considered programs is executed whenever the flow of control reaches this statement. There must not be any hidden condition not derived from the input-data preventing a statement from execution when it's reached by the flow of control.

Valuations of Variables

The set-theoretic formalism for describing structural code-coverage criteria and preservation conditions for structural code-coverage is based on valuations and sets of valuations of input-variables of a program [31].

A *valuation* of a variable is the assignment of a concrete value from its value domain to that variable. A vector $(v_1, \ldots, v_m)$ where v_i is a possible valuation of the i-th input variable of a program P is called *input-valuation* of P. The set of all possible input-valuations of a program P is denoted with $\mathbb{ID}$ and called the *input data* or *input data set* of that program.

A set $\mathbb{TD} \subseteq \mathbb{ID}$ defines the set of input-valuations selected for testing and is therefore called *test-data*. The case $\mathbb{TD} = \mathbb{ID}$ would mean exhaustive testing with test-data covering all possible input-data combinations. As described in Section 1.1 above, such a coverage is not a realistic goal due to the typically huge cardinality of $\mathbb{ID}$. So in the most common cases $\mathbb{TD}$ will be a true subset of $\mathbb{ID}$.

This thesis will assume, that a certain input valuation $id \in \mathbb{ID}$ will always trigger the execution of a specific path or scoped path. If a path triggered by an input-valuation id includes a statement x, then it is said, that *id triggers the execution of statement x*. The other way round, there may be several paths with one statement x in common, and therefore x can be triggered by more than one input-valuation.

Definition 2.1 (Reachability Valuation) *Let x be a simple statement, a basic block, a condition or a decision. Then the set $IV_R(x)$ denotes the set of all input-valuations that trigger the execution of x and $IV_R(x)$ is called* reachability valuation *of x.*

If x is a condition or a decision, then the *satisfyability valuation* defines the set of input valuations that trigger the execution of condition/decision x with a certain result of evaluating the logical expression of x.

Definition 2.2 (Satisfyability Valuation) *Let x be a condition or decision, then*

- *$IV_T(x)$ is the set of all input valuations that trigger the execution of x and the outcome of the evaluation result of x is* true.
- *$IV_F(x)$ is the set of all input valuations that trigger the execution of x and the outcome of the evaluation result of x is* false.

Due to the definition of $IV_T(x)$ and $IV_F(x)$ it is obvious, that the following relation between reachability- and satisfyability-valuation must always hold:

Corollary 2.1

$$IV_T(x) \cup IV_F(x) = IV_R(x)$$

Note, that in the common case the intersection $IV_T(x) \cap IV_F(x)$ is not the empty set. For example, consider a program structure where x is placed inside a loop and the evaluation of x depends on the iteration variable of that loop. Then it may happen, that two different iterations of the loop trigger different branches of x. Only if a condition or decision is addressed at most once in every execution of a program, then $IV_T(x) \cap IV_F(x) = \emptyset$ is true. From the point of view of the program structure that means, that the condition or decision must not be placed inside a loop or inside a subroutine that is called more than once.

Program Transformations

The transformation of a program P_1 into a program P_2 is denoted $P_1 \rightarrow P_2$. It means that P_2 is the final result of applying some well-defined transformation steps in a well-defined order to P_1 and its subsequent intermediate transformation results. Although some intermediate transformation steps may result in an incorrect program structure, P_1 and P_2 are required to be a valid program with respect to the chosen program representation. Program P_1 is often referred as the *original program* or *untransformed program* and P_2 is called the *transformed program*.

The individual steps of an optimizing transformation can involve reordering of program statements, replacing existing statements or inserting new code. This will cause changes in the sets of basic-blocks, conditions and decisions. Execution paths triggered by the input-valuations in a certain test-set $\mathbb{TD}$ may change and therefore the reachability of statements may differ compared with the original program. When test-data are generated from a higher-level representation of a program with the goal to achieve a certain kind of structural code-coverage the question arises, if the intended structural coverage will hold after transforming the higher-level representation of the program to object-code.

Conditions under which circumstances a program transformation preserves a certain kind of structural code-coverage are subject of the following sections and in [31]. These coverage preservation criteria are independent of a concrete test-data set and can be applied to any kind of code-transformations, like for instance on source-to-source transformations as well as on compilers. If it can be shown, that a program transformation preserves a certain kind of structural code-coverage then one can be sure, that the transformed program will achieve this certain kind of structural code-coverage for a given test-set $\mathbb{TD}$ whenever the untransformed program achieves this coverage criteria for the same test-set.

2.2 Statement Coverage

Statement Coverage (abbreviated SC) requires, that every statement in a program has been executed at least once with respect to a certain set $\mathbb{TD}$ of test-data. Achieving statement coverage also shows that every statement in the program is reachable. On the other hand, statement coverage is classified as a weak criterion, because it is insensitive to the control structure of the program [41].

Formally statement coverage is defined as follows [31]:

Definition 2.3 (Statement Coverage) *A set* $\mathbb{TD}$ *of test-data achieves* Statement Coverage *in a program* P *if the condition*

$$\mathbb{TD} \cap IV_R(b) \neq \emptyset$$

holds for all basic blocks $b \in B(P)$ *of that program.*

An aspect to consider is, that the classical view of statement coverage does not always concern conditions. But in most programming languages conditions may contain operational statements, often hidden behind function-calls, for example. So the use of operational statements as part of a condition can hide control flow inside logical expressions, if conditions are not considered for statement-coverage.

To determine, if a given program transformation preserves statement-coverage the following preservation condition can be used [31]:

Theorem 2.1 *Let* P_2 *be a program that results from transforming* P_1*. Then the transformation* $P_1 \rightarrow P_2$ *preserves statement coverage if and only if the following condition holds:*

$$\forall b' \in B(P_2)\ \exists\, b \in B(P_1) \text{ with } IV_R\left(b'\right) \supseteq IV_R(b)$$

In [31] a proof is maintained to show, that Theorem 2.1 is a necessary and sufficient condition for the preservation of statement coverage.

2.3 Decision Coverage

Decision Coverage (abbreviated DC), also called *Branch Coverage* or *Edge Coverage*, requires creating enough test cases such that each decision goes at least once into each possible direction [41]. The decision of each if-statement in a program for example, must evaluate at least once for the *true* and once for the *false* outcome of the decision to achieve decision coverage.

A formal definition of decision coverage is as follows [31]:

Definition 2.4 (Decision Coverage) *A set* $\mathbb{TD}$ *of test-data achieves* Decision Coverage *in a program* P *if the condition*

$$IV_T(d) \cap \mathbb{TD} \neq \emptyset \ \wedge\ IV_F(d) \cap \mathbb{TD} \neq \emptyset$$

holds for all decisions $d \in D(P)$ *of that program.*

The definition of decision coverage given in Definition 2.4 considers only two-way decisions and has to be modified for multi-way decisions like `switch`-statements in C/C++, for example.

Since every statement of a program is on some sub path starting at the programs entry point or at a branch decision, every statement will be executed at least once, if every branch direction is executed at least once. Therefore decision coverage implies statement coverage, but only if the program contains at least one decision [41].

Corollary 2.2 *Let P be a program with at least one decision. If a set $\mathbb{TD}$ of test-data achieves decision coverage in P, then it also achieves statement coverage.*

The preservation criteria for decision coverage (and also the preservation criteria for condition coverage below) uses a helper predicate $touches\_ID(x, ID)$, where ID is a set of input data and x can be a condition or decision. It is a check, whether some set of input data ID includes all of the *true*- or *false*-satisfyability valuation of a condition or decision x.

$$touches\_ID(x, ID) \ :\Longleftrightarrow \ (IV_T(x) \subseteq ID) \vee (IV_F(x) \subseteq ID) \tag{2.1}$$

Using the helper predicate (2.1), the preservation criterion for decision coverage is stated as follows:

Theorem 2.2 *Let P_2 be a program that results from the transformation $P_1 \rightarrow P_2$. The transformation preserves decision coverage if the following condition holds:*

$$\begin{aligned} \forall d' \in D(P_2) \qquad & \exists\, d \in D(P_1) \ : \ touches\_ID(d, IV_T(d')) \\ \textit{and} \qquad & \exists\, d \in D(P_1) \ : \ touches\_ID(d, IV_F(d')) \end{aligned}$$

The proof for Theorem 2.2 can be found in [31], showing that it is a necessary and sufficient criterion.

2.4 Condition Coverage

Condition Coverage (abbreviated CC) requires for each condition in a decision, that it take all possible outcomes at least once when the program is executed with a given set of test-data. This criterion does not require, that the decision itself takes all possible outcomes at least once [24]. For the decision $A \vee B$, for instance, the two test cases with $\langle A = true, B = false \rangle$ and $\langle A = false, B = true \rangle$ would satisfy condition coverage, but will cause the decision to take only one possible outcome.

The formal definition of condition coverage is similar to decision coverage with the difference, that it mentions conditions instead of decisions:

Definition 2.5 (Condition Coverage) *A set $\mathbb{TD}$ of test-data achieves* Condition Coverage *in a program P if the condition*

$$IV_T(c) \cap \mathbb{TD} \neq \emptyset \ \wedge \ IV_F(c) \cap \mathbb{TD} \neq \emptyset$$

holds for all conditions $c \in C(P)$ of that program.

Note, that special care is necessary if Boolean operators have a *shortcut semantics* (also called *short-circuit logic* [24]). Shortcut evaluation means, that the conditions remaining in the execution order sequence are not evaluated any more, if they cannot change the final outcome. For example, if the expression $A \vee B$ is evaluated from left to right, then the result of B could not change the final outcome if A evaluates to *true*. In a shortcut-semantics, execution of this

logical expression will therefore stop after A evaluates to *true*, because executing B seems to be superfluous.

Definition 2.5 requires, that each condition is really executed for each possible outcome [31]. Therefore, condition coverage in case of shortcut evaluation needs more test cases to assure that each possible condition outcome has been evaluated by executing the condition.

The preservation criteria uses again the helper predicate (2.1) and states as follows below [31]:

Theorem 2.3 *Let P_2 be a program that results from the transformation $P_1 \rightarrow P_2$. The transformation preserves condition coverage if the following condition holds:*

$$\begin{aligned} \forall c' \in C(P_2) \qquad & \exists\, c \in C(P_1) \;:\; touches\_ID(c, IV_T\left(c'\right)) \\ \textit{and} \qquad & \exists\, c \in C(P_1) \;:\; touches\_ID(c, IV_F\left(c'\right)) \end{aligned}$$

A proof for Theorem 2.3 can be found in [31], showing that it is a necessary and sufficient criterion.

2.5 Condition/Decision Coverage

Since decision coverage does not guarantee condition coverage and condition coverage does not guarantee decision coverage respectively, the obvious way out of this dilemma is to combine the requirements for decision coverage with the requirements for condition coverage. This criteria is called *Condition/Decision Coverage* (abbreviated CDC). It requires sufficient test cases that each condition of a decision takes each possible outcome at least once and that the decision goes in either direction at least once.

Definition 2.6 *A set $\mathbb{TD}$ of test-data achieves* Condition/Decision Coverage *in a program P if $\mathbb{TD}$ achieves decision coverage and also condition coverage.*

There is no special preservation criteria needed, because condition/decision coverage is preserved whenever condition coverage and decision coverage are preserved.

Corollary 2.3 *A transformation $P_1 \rightarrow P_2$ preserves Condition/Decision Coverage if it preserves decision coverage and condition coverage.*

Although it seems that condition/decision coverage exercises the effect of all possible outcomes of all conditions, it frequently does not [41]. Its weakness is, that certain conditions can mask other conditions and therefore hide the effect of changing one condition. In addition, not all combinations of conditions can be distinguished. The two test cases $\langle A = true, B = true\rangle$ and $\langle A = false, B = false\rangle$, for example, satisfy the condition/decision coverage requirement for the decision $A \vee B$, but they cannot distinguish it from the decision $A \wedge B$ [24].

2.6 Multiple Condition Coverage

Multiple Condition/Decision Coverage requires to test all possible combinations of condition outcomes in each decision. This is exhaustive testing of the input combinations of each decision. Although it may be a desirable criteria it is impossible in practice, since the number of possible combinations grows exponentially. Testing a decision with n conditions would require 2^n test cases to achieve multiple condition coverage [24].

2.7 Modified Condition/Decision Coverage

The *Modified Condition/Decision Coverage* (abbreviated MCDC) criterion was intended as a golden mean between multiple condition testing with its exponential growth of required test cases and condition/decision coverage testing. The essence of this criterion is, that each condition of a decision must be shown to independently affect the outcome of its decision. That means, one must demonstrate that the outcome of a decision changes as a result of changing a single condition while holding the outcome of all other conditions fixed. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. To achieve modified condition/decision coverage in a decision with n conditions, a minimum of $n+1$ test cases is needed [12, 24].

Showing, that a condition independently affects the outcome of a decision while holding all others fixed is commonly referred as the *unique cause approach*. If changing the value of a single condition changes the outcome of the decision, then the single condition is assumed to be the reason for that change. This can be formalized with the helper predicate $unique\_Cause$ defined as follows [31]:

$$\begin{aligned} unique\_Cause(c_1, d, td_1, td_2) \quad :\Longleftrightarrow \quad & control\_Expr(td_1, td_2, c_1) \\ & \text{and} \\ & control\_Expr(td_1, td_2, d) \\ & \text{and} \\ & \forall\, c_2 \in C(d) \text{ with } c_2 \neq c_1 \;\Rightarrow \\ & \quad is\_invariantExpr(\{td_1, td_2\}, c_2) \end{aligned} \tag{2.2}$$

The sub-predicate $control\_Expr$, used in (2.2) indicates, that the two test-data td_1 and td_2 are able to affect the outcome of a decision or condition x into both directions:

$$\begin{aligned} control\_Expr(td_1, td_2, x) \quad :\Longleftrightarrow \quad & td_1 \in IV_T(x) \wedge td_2 \in IV_F(x) \\ & \text{or} \\ & td_1 \in IV_F(x) \wedge td_2 \in IV_T(x) \end{aligned} \tag{2.3}$$

Finally, the sub-predicate $is\_invariantExpr$ of (2.2) indicates, that the provided test-data do not change the remaining conditions of the examined decision:

$$is\_invariantExpr(ID, x) \quad :\Longleftrightarrow \quad td_1 \cap IV_T(x) = \emptyset \quad \text{or} \quad td_2 \cap IV_F(x) = \emptyset \tag{2.4}$$

Using (2.2), modified condition/decision coverage can be formalized in the following manner:

Definition 2.7 (Modified Condition/Decision Coverage) *A set* $\mathbb{TD}$ *of test-data achieves* Modified Condition/Decision Coverage *in a program* P *if the condition*

$$\forall c \in C(d) : \exists td_1, td_2 \in \mathbb{TD} \quad \textit{such that} \quad unique\_Cause(c, d, td_1, td_2)$$

holds for all decisions $d \in D(P)$ *of program* P.

If the restriction that P has a single point of entry is dropped, then one must also require, that every point of entry is used at least once. This is the original definition of MCDC [12].

Note, that the definition of MCDC also subsumes condition- and decision-coverage.

Corollary 2.4 *If a set* $\mathbb{TD}$ *of test-data achieves* modified condition/decision coverage *in a program* P*, then it also achieves condition- and decision-coverage in* P *for the same set* $\mathbb{TD}$ *of test-data.*

The modified condition/decision coverage criterion is required for testing highly critical software in the avionics industry. The support for this testing is based on the assumptions, that a higher number of test cases will find more errors. Performing multiple condition coverage would be impractical, because in avionics systems, complex Boolean expressions are common [12, 24]. On the other hand, some authors argue that such highly complex decision do not really exist, because such complex expressions would make the code unreadable for the common practice of independent code-reviews. Therefore this explosion of test cases seems to be unfounded [9].

Developing a condition for preservation of multiple condition/decision coverage, a more general version of the helper predicate (2.3) with respect to the test-data set, is used:

$$\begin{aligned} mult\_control\_Expr(ID_1, ID_2, x) \quad :\Longleftrightarrow \quad & ID_1 \subseteq IV_T(x) \ \wedge \ ID_2 \subseteq IV_F(x) \\ & \text{or} \\ & ID_1 \subseteq IV_F(x) \ \wedge \ ID_2 \subseteq IV_T(x) \end{aligned} \tag{2.5}$$

The following preservation condition [31] for MCDC uses the sub-predicates (2.2) and (2.5) defined above:

Theorem 2.4 *Let* P_2 *be a program produced by a program transformation* $P_1 \rightarrow P_2$*. Then the program transformation preserves* modified condition/decision coverage *if and only if*

$$\forall\, d' \in D(P_2),\ \forall\, c' \in C(d') \quad \exists\, ID_1, ID_2 \subseteq \mathbb{ID}$$

such that the following condition holds:

$$\begin{aligned}
\exists\, d \in D(P_1),\ \exists\, c \in C(d),\ \exists\, ID_{tmp} \subseteq \mathbb{ID} \quad &: \quad multi_control_Expr(ID_1, ID_{tmp}, c) \\
&\textit{and} \\
\forall \langle id_1, id_2 \rangle \in ID_1 \times ID_{tmp} \quad &: \quad unique_Cause(c, d, id_1, id_2) \\
&\textit{and} \\
\exists\, d \in D(P_1),\ \exists\, c \in C(d),\ \exists\, ID_{tmp} \subseteq \mathbb{ID} \quad &: \quad multi_control_Expr(ID_2, ID_{tmp}, c) \\
&\textit{and} \\
\forall \langle id_1, id_2 \rangle \in ID_2 \times ID_{tmp} \quad &: \quad unique_Cause(c, d, id_1, id_2) \\
&\textit{and} \\
\forall \langle id_1, id_2 \rangle \in ID_1 \times ID_2 \quad &: \quad unique_Cause(c', d', id_1, id_2)
\end{aligned}$$

A proof for theorem 2.4, showing that it is a necessary and sufficient criteria, is given in [31].

The use of short-circuit operators may force a relaxation of the requirement to hold all other conditions fixed while the condition of interest is varying. Consider the expression $x \neq 0 \wedge y/x > 1$, for example. Short circuit evaluation will assure, that the second condition is never evaluated, if the first condition is *false*. In this case it is impossible to create test-cases where the first condition is held fixed to *false* and the second condition evaluates to either results, because y/x is undefined in this case. Another problem can arise if a decision owns at least two strongly coupled conditions, because it is not possible then to vary one condition and holding the other fixed. These cases need an alternative, less restrictive approach to MCDC [12, 31].

2.8 Path Coverage

Path coverage (abbreviated PC) requires, that test-data are selected such that each possible execution path of a program is executed at least once. Unfortunately experience shows that the number of execution-paths grows exponentially with the size of the program. Even small programs can have a possibly huge number of paths. If endless loops or endless recursions are possible, even the smallest program can comprise an infinite number of execution paths.

To reduce the number of test cases needed, paths that differ only by the number of iterations of a loop can be grouped together and then only a few representative paths from each group are executed. These restricted versions of path testing are known as *structured path testing* and *boundary interior path testing* [42].

2.9 Scoped Path Coverage

The idea behind *scoped path coverage* (abbreviated SPC) is to split a program P into hand able segments that are small enough to cover all possible paths inside each scope. To avoid the need of dealing with an infinite number of paths, programs with endless loops and endless recursion will be excluded by definition. Defining one program scope comprising the whole program P is just a special case of path-coverage [31].

The following definition 2.8 uses the expression $B_S(pp)$ to denote the basic block, the scoped path pp starts with. The expression $C_T(pp)$ denote the set of all conditions along the path pp, that evaluate to *true* and $C_F(pp)$ denotes the set of conditions along pp evaluating to *false*.

Definition 2.8 (Scoped Path Coverage) *A set* $\mathbb{TD}$ *of test-data achieves* scoped path coverage *in a program* P *if the condition*

$$\begin{aligned}\forall\, pp \in PP(ps) : \exists\, td \in \mathbb{TD} \quad \textit{such that} \quad & IV_R\left(B_S(pp)\right) \cap \{td\} \neq \emptyset \\ & \textit{and} \\ & \forall c_T \in C_T(pp) : IV_T\left(c_T\right) \cap \{td\} \neq \emptyset \\ & \textit{and} \\ & \forall c_F \in C_F(pp) : IV_F\left(c_F\right) \cap \{td\} \neq \emptyset\end{aligned}$$

holds for all program scopes $ps \in PS(P)$.

Partitioning a program into scopes is application specific and depends on the testing-goals. Scoped path coverage is a coverage metric introduced for the purpose of measurement based timing analysis. Measurement based timing analysis techniques combine static program analysis with execution time measurement to find a compromise between precision and safety on one side and the effort it takes on the other side [54, 60].

Defining a coverage preservation criteria for scoped path coverage [31] uses a helper predicate *is_CondTF_enclosed*, defined as follows:

$$\begin{aligned}is\_CondTF\_enclosed(ID, C_T, C_F) \quad :\Longleftrightarrow \quad & \exists\, c_T \in C_T : IV_T\left(c_T\right) \subseteq ID \\ & \text{or} \\ & \exists\, c_F \in C_F : IV_F\left(c_F\right) \subseteq ID\end{aligned} \tag{2.6}$$

Using (2.6) the preservation condition for scoped path coverage is stated as follows:

Theorem 2.5 *A program transformation* $P_1 \rightarrow P_2$ *preserves* scoped path coverage *if the condition*

$$\begin{aligned}\forall pp' \in PP(ps') \quad : \quad & \exists\, ps \in PS(P_1),\ \exists pp \in PP(ps) \quad \textit{such that} \\ & IV_R\left(B_S(pp')\right) \supseteq IV_R\left(B_S(pp)\right) \\ & \quad \textit{and} \\ & \forall\, c' \in C_T(pp') \quad \exists\, ps \in PS(P_1) \quad \exists\, pp \in PP(ps) \quad \textit{such that} \\ & \quad is\_CondTF\_enclosed(IV_T\left(c'\right), C_T(pp), C_F(pp)) \\ & \quad \textit{and} \\ & \forall\, c' \in C_F(pp') \quad \exists ps \in PS(P_1) \quad \exists pp \in PP(ps) \quad \textit{such that} \\ & \quad is\_CondTF\_enclosed(IV_F\left(c'\right), C_T(pp), C_F(pp))\end{aligned}$$

holds for all scoped paths $ps' \in PS(P_2)$ *of the transformed program.*

A proof, showing that theorem 2.5 is a necessary and sufficient condition for preservation of scoped path coverage is given in [31].

2.10 Summary of Chapter 2

Basic terms and definitions are introduced in this chapter as a foundation to describe programs, program transformations and structural code coverage in a formal way. An informal survey of several code-coverage criteria is given. For certain structural code-coverage criteria formal definitions and formal conditions for their preservation are explained.

CHAPTER 3

A Program-Model for Coverage Preservation Analysis

This chapter establishes the formalism, automatic coverage preservation analysis is based on. Section 3.1 starts with definition of a control-flow graph model as representation language for programs. Section 3.2 describes some basic properties of the control-flow graph model. Sections 3.3 and 3.4 continue to describe properties of program transformations. Finally, Section 3.5 is putting all together and presents a principle procedure for analysis and some examples for analyzing the preservation of statement coverage, condition coverage and decision coverage.

3.1 Definition and Examples

Introduction

The code-coverage-analysis procedure presented in this thesis is based on a control-flow graph model [4] for programs or fragments of a program. Control-flow graphs are well established in discrete event simulation and in different types of control-flow analysis and data-flow analysis(see, e.g., [48, 13, 26]). They are very illustrative objects and in most cases their structure can be derived directly from the program under investigation.

A *control-flow graph* (abbreviated CFG) is a kind of transition system. It uses a directed graph, also called *digraph* [11], to express the basic control-flow relationships in a program or in a fragment of a program. Every vertex in a control-flow graph represents a basic block of code. Directed edges represent possible transfers of the flow of control between these blocks.

The CFG-model used in this thesis will establish some enhancements specifically tailored for code coverage analysis. These additions will also support the implementation of the analysis technique in a mathematical software system. This thesis uses the term *analysis control-flow graph* (abbreviated *aCFG*) to distinguish the enhanced CFG for code-coverage analysis from classical control-flow graphs.

Compared with classical CFG-models like the one described in [4], the aCFG used in this thesis for coverage preservation analysis will provide a higher grade of detail. Its structure is based explicitly on single statements or basic groups of single statements, whereas classical control-flow graphs use basic blocks. In addition, a supplementary set is introduced to define the structures of the programs decisions. Finally, the edges in an aCFG will carry additional information allowing tracking the correlation between input-valuations and execution paths.

To meet the restrictions described in Section 2.1, this thesis will require that programs and fragments of programs under investigation have single points of entry and exit. Therefore, an aCFG as a representation of such a program must have a well-defined single entry node and a single exit node. Furthermore, it will be assumed that the statement, represented by a node of an aCFG will be executed without further conditions, as soon as the flow of control reaches that node.

Statements and Nodes

If an aCFG represents a program or a fragment of a program, then each node of the aCFG represents a simple statement, a statement sequence or a condition. However, the mathematical structure of a node does not depend on the type of statement it represents. The distinction between different types of statements results from how edges link nodes together.

A single node with one outgoing edge is called a *simple node* and represents a *simple statement*, because a simple statement includes no control-flow decision. A simple statement has therefore exactly one successor in every execution sequence that includes this statement when executing the program or program fragment P. Simple statements may have more than one predecessor and therefore the node representing a simple statement in an aCFG may also have an arbitrary number of incoming edges.

In most cases it is convenient to unite a linear sequence of simple statements to one object. A linear sequence of simple statements is a sequence $s_1, s_2, \ldots, s_r$ such that for all $i = 2 \ldots r$ always s_{i-1} is the only one predecessor of s_i. Such a sequence is called a *statement sequence* and can be treated like a super-instruction. Since there are no joins and forks allowed inside a statement sequence, execution of the first statement of a sequence implies that all statements of the sequence are executed in order.

In an aCFG a simple node represents a statement sequence. The input of a statement sequence is the input of the sequences first simple statement and the output is the result of executing all statements of the sequence exactly in order. The predecessors of a statement sequence are the predecessors of the sequences first simple statement. The successor is the successor of its last simple statement. Note, that statement sequences are not required to be maximal. So the length of the sequence can be chosen specific to the application.

A node with more than one outgoing edges is called a *condition node* and represents a *condition*. A condition includes a control-flow decision with a predefined set of possible successor statements. The successor where the flow of control continues is determined by the result of the conditions execution. In general, conditions may have an arbitrary number of results and many programming languages provide such multiple-branch statements. The `switch`-statement in C/C++, for instance, branches according to the actual value of an integer-variable and can therefore produce a large number of possible outcomes. However, as already mentioned before this

thesis will only deal with conditions producing Boolean results and therefore having two possible successors.

The mathematical structure of a node of an aCFG is a composition of two mandatory and one optional element. The mandatory elements are a tag taken from an arbitrary tag set and a unique node identifier taken from an identifier set. As an optional element, the node definition may contain a related-node identifier also taken from an identifier set.

Definition 3.1 *Let N and N' be two sets of identifiers and let Λ be an arbitrary set of tags. Then a*

tagged node *is a tuple $v = \langle \lambda, n \rangle$ with $\lambda \in \Lambda$ called the* tag *of the node and $n \in N$ being the nodes* identifier.

related tagged node *is a tuple $v = \langle \lambda, n, n' \rangle$ with $\lambda \in \Lambda$ called* tag *of the node, $n \in N$ being the nodes unique* identifier *and with an identifier $n' \in N'$ called the nodes* related identifier.

Node identifiers serve as unique identification of the nodes of an aCFG. Within this thesis, node identifiers are taken from a set of integers. In contrast to node identifiers, node tags are not required to be unique. At the moment, node tags are only used for documentary reasons without functional purpose. This thesis will use text-strings describing the function of the statement represented by a node.

If an aCFG P_2 is the result of a transformation $P_1 \rightarrow P_2$, then related tagged nodes may be used for some nodes of the aCFG P_2. The related-node identifier $n' \in N'$ of a node $n \in N$ of P_2 is a designation for the node of the transformed program and refers to the instance of that statement in the original program P_1. In this case, the set N' is the set of the node identifiers of P_1 and N comprises the node identifiers of P_2. In other words, the related-node identifier is a pointer from a statement n of the transformed program to a statement n' of the original program, if the node n' was used by the program transformation to construct the node n of the transformed program. Constructing a node of the transformed program may include any kind of action, ranging from using a one-by-one copy of the original statement up to creating a new sequence of statements replacing the original statement. More than one statement of the transformed program may refer to the same statement in the original program, if one statement is used several times for creating statements of the transformed program. However, related-tagged nodes and tagged nodes may be used together in the same aCFG representing different statements, since not every statement of a transformed program needs to be created on basis of a statement of the original program.

In conformance with the definitions in Section 2.1, the symbol $B(P)$ will be used to refer to the set of nodes representing simple statements or statement sequences of a program P. The node-set representing conditions of P will be denoted $C(P)$. In addition, $V(P)$ represents the unified node set $B(P) \cup C(P)$, and finally $ST(P)$ denotes the tuple $\langle s, t \rangle$, where s is the entry node and t is the exit node of the aCFG representing the program P.

Decisions

The decision set of a program or a program fragment P defines a partition of the programs set of conditions. In the aCFG representing P the sub-graph representing a decision is treated as a *hyper-node* [23] grouping together the condition nodes the decision is composed of. Similar to single nodes, the formal definition of a decision comprises also a tag used as an additional identification. This thesis will use text strings describing the branch statement the decision hyper-node is representing. For example, a decision hyper-node is named "while" if it represents the loop decision of a *while* loop. As described for single nodes, the tag of a decision mainly has documentary functions. In addition, the framework implementation for automatic analysis uses the decision tag to recognize loop-control decisions. This implementation specific feature is described in Section 4.1.

Definition 3.2 *Let P be a program and $\lambda_i \in \Lambda$ a tag from an arbitrary tag set Λ. Also let $\{c_{i_1}, \ldots, c_{i_k}\} \subseteq C(P)$ be a subset of the programs condition set.*

A **tagged decision** *of P is a tuple $d_i := \langle \lambda_i, c_{i_1}, \ldots, c_{i_k} \rangle$, and λ_i is called the* decision tag *of d_i and $\{c_{i_1}, \ldots, c_{i_k}\}$ is called the* condition set *of d_i.*

Note, that the order of the conditions inside the decision has no meaning and can be chosen arbitrary. As defined in Section 2.1, $D(P)$ denotes the set of decisions of a program P and $C(d_i) = \{c_{i_1}, \ldots, c_{i_k}\}$ denotes the set of conditions the decision $d_i \in D(P)$ is composed of.

For a decision d_i any edges with both endpoints being part of $C(d_i)$ are called *internal edges* of the decision. If one endpoint of an edge e is member of $C(d_i)$ while the other endpoint is not, e is called an *external edge* of the decision. Internal edges are relevant for defining how the conditions are logically linked together composing the decision while the external edges are relevant for linking a decision to other nodes or hyper-nodes of an aCFG.

The notion of terms for single nodes is applied by analogy to the hyper-nodes representing decisions. For instance, in analogy to single nodes an external edge ending inside a decision is called incoming edge and the number of incoming edges is called incoming degree of a decision hyper-node.

Control Flow Edges

Control flow edges inside an aCFG represent the transition relation between the statements of a program P.

Definition 3.3 *Let $V(P)$ be the set of nodes representing all statements of a program P, let $\mathbb{ID}$ be the set of possible input data for P, and let $\{\alpha_1, \ldots, \alpha_r\}$ a set of arbitrary markers.*

A **labelled control flow edge** *e of an aCFG P is a directed edge represented by a tuple $e := \langle v, w, \alpha, \delta \rangle$ with the following components:*

1. *A pair v, w with $v, w \in B(P) \cup C(P) \cup ST(P)$, defining the endpoints and the direction of the edge. The node v is the origin of the edge and is called* head *of the edge, w is the destination of the edge and called* tail *of the edge.*

2. *A component* α*, called the* condition/decision label *of* e*.* α *can be empty or an element of the set* $\{true, false\} \times \{X, true, false, \alpha_1, \ldots, \alpha_r\}$.

3. *A set of input valuations* $\delta \subseteq \mathbb{ID}$*, called the* input-valuation set *of* e*.*

Head and tail of an edge e are denoted $head(e)$ and $tail(e)$ [22]. If two nodes are connected with multiple edges, then they are required to differ in their condition/decision label.

Figure 3.1: Components associated with an edge e of an aCFG representing a program P, dependent on the type of statement represented by $head(e)$.

The *condition/decision label* of an edge e is empty, if $head(e)$ represents a simple statement. In case of a condition the condition/decision label is used to mark the edge where control flow continues depending on the result of evaluating the condition represented by $head(e)$. The condition/decision label comprises two parts, one for the condition, called *condition label* and one for the decision, called *decision label*.

The condition label is a marker for the outcome of the condition represented by $head(e)$. It can either be *true* or *false* and determines the edge, where the flow of control continues if the outcome of the condition represented by $head(e)$ evaluates to the corresponding result. For instance, if the expression represented by $head(e)$ evaluates to *true*, flow of control continues using the edge marked with condition label *true*. If more than one edges with the same condition label leaves a node, the edges must differ in their decision label. On the other hand, a condition-node must have at least two outgoing edges, one for *true* and one for *false*.

The decision label represents the possible results of the decision, so far as it can be predicted after the evaluation of $head(e)$. Evaluating the result of a decision is a lot more complex than evaluating the result of single conditions, because the outcome of a decision can depend on the result of several conditions. Therefore, the number of possible markers is higher than for conditions. In the simplest case, the result of a decision is finalized after executing a certain condition. In this case, the decision label is either *true* or *false*. On the contrary, the result of the decision can be completely undetermined after evaluation of a certain condition. In this case, the symbol X is used to express the fact, that the result is still undecided. Obviously, if the outcome of a decision is fixed on a certain point, the result must not be changed on any further feasible path through the decision.

For example, think of a decision $A \wedge B$, with A and B being two conditions evaluated from left to right. If A evaluates to *false*, then the result of the decision is fixed to *false* and can't be changed by the result of B. In the other case, when A evaluates to *true*, the final result of the decision can be either *true* or *false*, dependent on the result of evaluating B.

In some cases, the result after evaluating a condition is still undetermined, but with the tendency to a certain result after evaluating the rest of the decisions logical expression. For instance, think of the logical expression $A \Leftrightarrow B$ and again evaluate it from left from right. Although the result of the decision is not fixed after evaluating A, the result of A has an influence on how B affects the final result. To handle such cases, arbitrary markers can be used to mark feasible paths through a decision, if the result of a condition sets up a tendency for the result of the whole decision. Nevertheless, external edges of a decision must be marked in a way that implies an unambiguous final result. If arbitrary markers are used on external edges of a decision, they must be clearly related to a result of *true* or *false*.

The component δ is a set of input valuations attached to an 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 elements of $\mathbb{ID}$. A more detailed description, how the input-valuation sets attached to the control-flow edges of an aCFG support the analysis follows in Section 3.2.

Figure 3.1 introduces a notation used for drawing a graphical representation of an aCFG. For several reasons an abbreviated string notation is used for the condition/decision labels instead of a tuple-notation. The character on the left-hand side of such strings is always a condition label and the character on the second position is reserved for the decision label. For both labels *true* will be abbreviated with the letter "T" and *false* will be denoted using the letter "F". The letter "X" is the symbol for unknown decision result. The string "TT", for instance, is an abbreviation for the condition/decision label $\langle true, true \rangle$.

Analysis-Control-Flow Graph (aCFG)

Definition 3.4 (Analysis-Control-Flow Graph (aCFG)) *A directed connected graph defined by the tuple $P = \langle B, C, D, R, s, t \rangle$ is called* analysis-control-flow graph *(aCFG) representing a program or a program fragment if it comprises the following components with the following properties:*

B *is a set of tagged nodes or related tagged nodes with unambiguous identifiers, representing the simple statements or statement sequences of a program. It is called the* statement set *of P.*

C *is a set of tagged nodes or related tagged nodes with unambiguous identifiers, representing the conditions of a program and called the* condition set *of P.*

D *is a set of tagged decisions forming a partition of C, called the* decision set *of P.*

R *is a set of aCFG edges over $B \cup C \cup \{s, t\}$ correlated to the programs input-data set $\mathbb{ID}$, called the* control flow edges *of P.*

s,t *are two tagged nodes or related tagged nodes not part of B or C with unambiguous identifiers, representing the programs unique nodes of entry and exit. The node s is called the* start node *or* entry node *of P and t is called the* terminal node *or* exit node *of P.*

Note, that in the rest of this thesis an aCFG will be used as abstraction for a program, preserving the details of the program as far as it is necessary for the analysis. As described on Section 2.1, the notation $B(P)$, $C(P)$, $V(P)$ and $D(P)$ will be used to identify the different node sets of a program P. In addition, the symbol $R(P)$ denotes the set of edges of P and $ST(P)$ denotes the tuple $\langle s, t\rangle$ containing the entry node and the exit node of P.

Since the decisions of a program P are defined to create a partition over $C(P)$, the condition sets $C(d_i)$ of all decisions $d_i \in D(P)$ must fulfil the same properties as blocks of a set-partition:

$$\text{(i)} \quad C(d_i) \cap C(d_j) = \emptyset \qquad \forall\, i \neq j \tag{3.1}$$

$$\text{(ii)} \quad \cup_i C(d_i) = C(P) \tag{3.2}$$

In other words, two different decisions must not share any condition and each condition must be part of some decision. Note also, that the current definition does not allow the presence of multiple instances of one condition inside a decision and therefore the model does not support coupled conditions (ref. to Page 8).

Further Definitions

An *execution path* represents a possible sequence of statement executions of the program represented by an aCFG P. Paths are defined by sequences of edges rather than sequences of nodes. The background of this choice is, that the essential information for coverage analysis is associated with edges, and the nodes are identifiable by the common end-points of succeeding edges.

Definition 3.5 (Execution Path) *Let P be an aCFG with entry-node s and exit-node t, and v, w be two arbitrary nodes of P. Let $pp := \langle e_1, \ldots, e_r\rangle$ be a sequence of edges with possibly multiple occurrences of the same edge. Then*

1. *The sequence pp is called* execution path from v to w *if $head(e_1) = v$, $tail(e_r) = w$ and if the equality $tail(e_i) = head(e_{i+1})$ holds for each $1 \leq i \leq r-1$.*
2. *If $v = s$ and $w = t$ the sequence pp is called* execution path through P *or simply* execution path *of P.*

Note, that the graph-theoretic definition of a *path* typically requires, passing each edge at most once. In contrast to that, Definition 3.5 allows multiple instances of every edge. In graph theory this is often addressed as *walk* through the graph [22].

As described in Section 2.1, the set of all execution paths of an aCFG P is denoted $PP(P)$. The path triggering function $path : \mathbb{ID} \rightarrow PP(P)$ is a surjection, since typically more than one input valuation will trigger the same path. On the other hand it is assumed, that $PP(P)$ only contains such executions paths, which are triggered by at least one input valuation of the program. This unambiguous trigger relation includes some important assumptions.

- All input data are fixed when program execution starts and there will be no additional input during execution.
- The execution path is determined in a predictable way and there are no random variables with influence on any decision inside the program.

The next definition establishes a CFG-like equivalence for scopes of a program. The idea behind this definition is to pick out some arbitrary detail of an aCFG.

Definition 3.6 *A* scope *or* aCFG-scope *of an aCFG P is a tuple $H := \langle V, E \rangle$ with the following characteristics:*

1. $V \subseteq B(P) \cup C(P)$
2. $E = \{e \in R(P) \mid head(e) \in V \vee tail(e) \in V\}$

If $head(e) \in V$ and $tail(e) \in V$ for some $e \in E$, e is called internal edge *of H. If for some $e \in E$ the condition $head(e) \notin V$ or $tail(e) \notin V$ is true, e is called* external edge *of H.*

The set of all possible aCFG-scopes of an aCFG P will be denoted with $PS(P)$ in conformance with the definitions in Section 2.1. Note, that the smallest possible scope consists of a single node with its incoming and outgoing edges.

Be aware, that a scope is neither a valid aCFG, nor a valid graph in general, because it may contain edges with one endpoint not member of the node set of the scope. It has neither a single entry point nor a single exit point, and it does not require including all conditions of a decision hyper-node. A scope of an aCFG should be seen as a focus on some details of an aCFG, like looking through a spyglass.

Examples

The examples in this section will illustrate the application of the aCFG model. It presents two possible implementations of a branch-statement. The investigated snippet of the program is written down in Figure 3.2(a) using a pseudo-code language. The branch-decision is composed of two conditions linked by a Boolean *and*-operator. If the decision evaluates to *true*, a statement-sequence $s1$ should be executed. If the result is *false*, execution continues with statement-sequence $s2$.

The first version of the aCFG shown in Figure 3.2(b) demonstrates an example of shortcut-evaluation semantics. Shortcut evaluation is used in some programming languages to optimize the evaluation of Boolean expressions. It operates in such a way, that 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 $\langle 2, 5 \rangle$ to continue control-flow immediately with statement sequence $s2$ if execution of condition A results *false*.

In contrast Figure 3.2(c) implements the same piece of code but without shortcut-evaluation semantic. After executing condition A execution continues evaluating condition B. Although there is only one execution path, two edges are needed to express the fact, that node 2 represents a

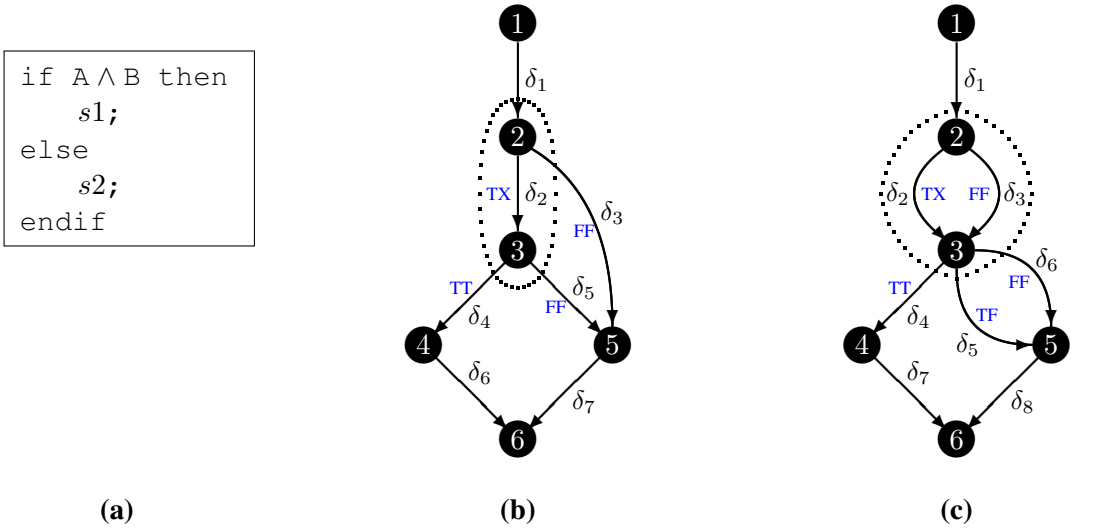


Figure 3.2: A branch statement and two possible representations by an aCFG with and without shortcut-evaluation semantics.

condition that can evaluate to either *true* or *false*. In contrast to the shortcut version, continuation of the control flow after evaluating B does not only depend on the local result. Only if B evaluates to *false* then execution will always continue with the *else* branch. But in case of result *true* the control flow may continue in either directions dependent on the result of A. Therefore two edges for condition result *true* with different decision results are necessary to model the difference in the decision result.

Integer-numbers serve as unique identifiers of the aCFG nodes. For the tags of the nodes text strings are chosen, describing the function of the program statement represented by a node. In addition, the entry node has identifier 1 and is tagged with "start" and the exit node 6 is tagged with "end". Input-valuation sets are denoted using the Greek letter δ with an index.

Since both graph representations use the same symbols for the same objects, the formal definition of the node sets and edge sets are quite similar. Node 4 and 5, representing the operations inside the *then* and *else* branch, are the only simple statements or statement sequences. They are therefore members of the statement set B. The condition-nodes (identifiers 2 and 3) are members of the condition-set C. Both condition nodes are forming the only one decision $\langle 2, 3\rangle$ of this example and the only one member of the set D. In the graph, they are enclosed by a dotted line to emphasize the hyper-node representing the decision. Therefore, the sets B, C, D and the special nodes s and t are defined as follows:

$$
\begin{aligned}
B &= \{\langle \text{“}s1\text{“}, 4\rangle, \langle \text{“}s2\text{“}, 5\rangle\} \\
C &= \{\langle \text{“}A\text{“}, 2\rangle, \langle \text{“}B\text{“}, 3\rangle\} \\
D &= \{\langle \text{“}if\text{“}, 2, 3\rangle\} \\
s &= \langle \text{“}start\text{“}, 1\rangle \\
t &= \langle \text{“}end\text{“}, 6\rangle
\end{aligned}
$$

The obvious difference of the two aCFGs is how nodes are linked together. The edge sets for both variants of the program are listed below.

Shortcut version, figure 3.2(b):

$$R_s = \{ \langle 1, 2, \text{“”}, \delta_1 \rangle, \langle 2, 3, \text{“}TX\text{“}, \delta_2 \rangle, \langle 2, 5, \text{“}FF\text{“}, \delta_3 \rangle, \langle 3, 4, \text{“}TT\text{“}, \delta_4 \rangle, \langle 3, 5, \text{“}FF\text{“}, \delta_5 \rangle, \langle 4, 6, \text{“”}, \delta_6 \rangle, \langle 5, 6, \text{“”}, \delta_7 \rangle \}$$

Non-shortcut version, figure 3.2(c):

$$R_f = \{ \langle 1, 2, \text{“”}, \delta_1 \rangle, \langle 2, 3, \text{“}TX\text{“}, \delta_2 \rangle, \langle 2, 3, \text{“}FF\text{“}, \delta_3 \rangle, \langle 3, 4, \text{“}TT\text{“}, \delta_4 \rangle, \langle 3, 5, \text{“}TF\text{“}, \delta_5 \rangle, \langle 3, 5, \text{“}FF\text{“}, \delta_6 \rangle, \langle 4, 6, \text{“”}, \delta_7 \rangle, \langle 5, 6, \text{“”}, \delta_8 \rangle \}$$

Finally, the complete aCFG definitions P_s for the shortcut version and P_f for the full evaluated version of the conditional branch look as follows:

For the shortcut version figure 3.2(b):

$$P_s = \langle B, C, D, R_s, s, t \rangle$$

For the non-shortcut version, figure 3.2(c):

$$P_f = \langle B, C, D, R_f, s, t \rangle$$

Please be aware, that the choice of tags, node-identifiers, and input-valuation set symbols is free, as long as it meets the limitations of unambiguity stated in Definition 3.4. So the presented aCFG representations for the piece of code in Figure 3.2(a) are only one of many possibilities. Only the principle structure of the graph is independent of naming and drawing issues, because it reflects the structure and the semantics of the program.

3.2 Input-Valuation Relations

This section introduces basic relations between the input-valuation sets of adjacent edges of a node. This is possible, although neither any assumptions have been made about the input-data set $\mathbb{ID}$, nor any knowledge is available about the elements of $\mathbb{ID}$. The principle for gaining 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* [2]. 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. The foundation for this similarity is the fact, that each input valuation triggers a path sourcing at a specific entry node and terminating at a specific exit node. And as well as that, an execution path must form a feasible continuous sequence of statements according to the programs transition relation.

Basic Properties

The foundation for the relations between input-valuation sets associated with the edges of the aCFG is expressed by two axioms forming the heart of the valuation analysis. Axiom 3.1 formalizes the properties of a single-entry point and a single-exit point of an aCFG while Axiom 3.2 will establish a property of continuity for input valuations inside an aCFG.

Axiom 3.1 *If P is an aCFG with entry-node s and exit-node t. Then the following properties are true:*

1. *s has no incoming edges and exactly one outgoing edge, and s is the only one node of this kind.*
2. *t has an arbitrary number of incoming edges but no outgoing edges, and t is the only one node of this kind.*
3. *Let δ denote the input-valuation set associated with the outgoing edge of s and let ϑ_i be an input-valuation set associated with an edge $e_i \in R(P)$. Then the relationship $\delta \supseteq \vartheta_i$ holds for all i.*

Axiom 3.2 (Conservation Axiom) *Let $\delta_1, \ldots, \delta_{m_k}$ denote the input-valuation sets associated with the incoming external edges of an arbitrary aCFG scope ps_k and $\varrho_1, \ldots, \varrho_{n_k}$ denoting the input-valuation sets associated with the external outgoing edges of the same scope ps_k, then the following relations are true for every scope $ps_k \in PS(P)$:*

1. *The input-valuation sets associated with the incoming-external edges and the outgoing-external edges of ps_k fulfil the equality*

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

2. *For each input-valuation set ϑ associated with some internal edge of ps_k*

$$\bigcup_{i=1\ldots m_k} \delta_i \supseteq \vartheta$$

The entry node and the exit node are acting as an interface to the environment of the program or program fragment represented by the aCFG. To serve this function the entry node "produces" the input valuations entering the aCFG from its environment and the exit node "consumes" the input valuations handed over to the environment. If the aCFG represents a complete program, the input-valuations "produced" by s are the input-valuations of the program. If the aCFG represents a fragment of a program, the input-valuations "produced" by s are restricted to the subset of the programs input data triggering paths through this fragment.

Part (1) and (2) of Axiom 3.1 require, that the entry node and the exit node exclusively act as an interface. But over and above they are not involved into the actual program structure. For example, the entry node or the exit node is not allowed to be part of a loop structure. The

essential statement of part (3) of Axiom 3.1 is, that any path through the aCFG must start at the entry node. Axiom 3.2 is a statement about the properties of the inner nodes of an aCFG. It implies, that the inner nodes must not produce or consume any input valuations. This property is comparable to the *Kirchhoff law* in the theory of electrical networks.

Together both axioms are a formal description for the assumption, that each execution path of P originates at the entry node, terminates in the exit node, and that there must not be any gaps in between. Using the graph-theoretic idea established with Definition 3.5, a path must always form an unbroken sequence of edges where all input-valuations triggering that path, must be always member in the associated input-valuation set of each edge.

Axiom 3.1 together with Definition 3.5 also has a consequence, that will be important for the analysis:

Corollary 3.1 *Let δ denote the input-valuation set associated with the outgoing edge of the entry node s and $\varrho_1, \ldots, \varrho_n$ denoting the input-valuation sets associated with the incoming edges of the exit-node t. Then*

$$\delta = \bigcup_{i=1\ldots n} \varrho_i$$

This fact is directly implied by Axiom 3.2, if a scope comprising all nodes $B(P) \cup C(P)$ is chosen. An important consequence of Corollary 3.1 is, that non-terminating loops inside the aCFG are not allowed, because these would violate the input-output equality of some scope enclosing the loop.

Remarks: This thesis will only deal with entry-nodes having a single outgoing edge. Multiple outgoing edges may be a possible enhancement for the future to handle optimizations using parallel execution on different hardware entities. With exception of the empty program, direct edges from the entry-node to the exit-node will not be allowed, because they make no contribution to the analysis.

Local Input-Valuation Relations

Applying the basic properties of input-valuation relations described above to certain predefined scopes of the aCFG, it is easy to determine several non-strict superset relations between input-valuation sets associated with the edges of the aCFG. Obtaining such non-strict superset relations is an important preparation step for the analysis, because nearly all of the conditions that must be fulfiled when checking preservation of a certain code coverage criteria, are based on non-strict superset relations between some sets of input-valuations. The non-strict superset relations will be called *input-valuation relations* for short and input-valuation relations inside one aCFG are called *local input-valuation relations* or *local relations*.

For finding local input-valuation set relations, corresponding to Axiom 3.2 it would be necessary to pick-up every possible scope of the program. Then for each such scope, a bundle of input-valuation relations can be obtained as described below. In the following description $\delta_1, \ldots, \delta_m$ denotes the input-valuation sets associated with the incoming external-edges of the inspected scope, and $\varrho_1, \ldots, \varrho_n$ denotes the input-valuation sets associated with the outgoing external edges of the inspected scope.

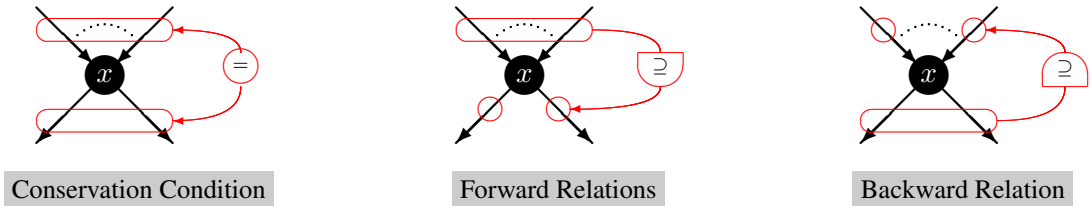


Figure 3.3: Determining local input-valuation relations based on the conservation axiom.

1. The equality $\delta_1 \cup \ldots \cup \delta_m = \varrho_1 \cup \ldots \cup \varrho_n$ can be directly derived from the conservation axiom.

2. Using the equality above, the following forward relations can be derived by set-theoretic considerations:

   $$\delta_1 \cup \ldots \cup \delta_m \supseteq \varrho_i \qquad 1 \leq i \leq n$$

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

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

Figure 3.3 illustrates the principle of obtaining a set of local input-valuation relations using an example with a scope that is restricted to one node.

The basic relations mentioned so far only consider input-valuation sets associated with edges incoming to and outgoing from the same scope. But using the transitivity of the superset-relation

$$\delta \supseteq \vartheta \quad \text{and} \quad \vartheta \supseteq \varrho \quad \Longrightarrow \quad \delta \supseteq \varrho$$

the set of relations can be expanded to input-valuation sets associated with edges not adjacent to one scope.

When developing a handsome procedure for determining local input-valuation relations, inspecting every possible scope of an aCFG can become a very complex task, even for small aCFGs. To avoid this effort the set of local input-valuation set relations will be restricted using only obvious scopes immediately available. Such obvious scopes are the nodes $v \in B(P) \cup C(P)$ (all nodes without entry node and exit node), and the hyper-nodes $d \in D(P)$ representing the decisions of the inspected program. So, walking through each node and hyper-node of the aCFG reveals a bundle of local relations that can serve as basis for code-coverage preservation analysis.

Example

As an example consider the representation of the branch-statement with shortcut-evaluation presented in figure 3.2(b) on page 27. Inspecting the nodes $2, 3, 4, 5$ reveals the following valuation-

relations:

$$
\begin{aligned}
\text{Node } 2 \quad &\Rightarrow \quad \delta_1 = \delta_2 \cup \delta_3 \\
&\qquad\quad \delta_1 \supseteq \delta_2 \\
&\qquad\quad \delta_1 \supseteq \delta_3 \\
\text{Node } 3 \quad &\Rightarrow \quad \delta_2 = \delta_4 \cup \delta_5 \\
&\qquad\quad \delta_2 \supseteq \delta_4 \\
&\qquad\quad \delta_2 \supseteq \delta_5 \\
\text{Node } 4 \quad &\Rightarrow \quad \delta_4 = \delta_6 \\
\text{Node } 5 \quad &\Rightarrow \quad \delta_3 \cup \delta_5 = \delta_7 \\
&\qquad\quad \delta_7 \supseteq \delta_3 \\
&\qquad\quad \delta_7 \supseteq \delta_5
\end{aligned}
$$

Corollary 3.1 finally creates a relation between the entry node and the exit node of the investigated aCFG:

$$
\begin{aligned}
&\Rightarrow \quad \delta_1 = \delta_6 \cup \delta_7 \\
&\qquad\quad \delta_1 \supseteq \delta_6 \\
&\qquad\quad \delta_1 \supseteq \delta_7
\end{aligned}
$$

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).

Remark: Because of the equality in case of simple statements like $\delta_4 = \delta_6$ it would be correct to associate the same input-valuation set symbol with both edges instead of using different symbols. In the example it would be possible to associate δ_4 with the edges $\langle 3, 4\rangle$ and $\langle 4, 6\rangle$. But for practical reasons it is highly recommended to use different input-valuation set symbols for different edges. This supports keeping track of what is going on during coverage preservation analysis.

Reachability and Satisfyability Valuation in aCFGs

Reachability valuation and satisfyability valuation from Definitions 2.1 and 2.2 can be transposed to the aCFG model in an obvious way. The reachability-valuation set of a node is determined by "summing up" the input-valuation sets on all edges incoming to a node or hyper-node. Each satisfyability valuation is determined in an analogous way by "summing up" all input-valuation sets on all outgoing edges marked with a certain condition label or decision label.

Definition 3.7 *Let P be an aCFG representing a program.*

1. *If x is a node or hyper-node of P representing a simple statement, a statement-sequence, a condition or a decision and $\delta_1, \ldots, \delta_m$ denotes the input-valuation sets associated with*

the incoming edges of x, then

$$IV_R(x) = \bigcup_{i=1\ldots m} \delta_i$$

2. *If x is a node of P representing a condition or a hyper-node representing a decision, and if $\varrho_1, \ldots, \varrho_r$ denotes the valuations-sets associated to the outgoing edges of x with condition label or decision label* true *and $\varrho_{r+1}, \ldots, \varrho_n$ denotes the input-valuation sets associated to the outgoing edges of x with condition label or decision label* false. *Then*

$$IV_T(x) = \bigcup_{i=1\ldots r} \varrho_i \quad \textit{and} \quad IV_F(x) = \bigcup_{i=r+1\ldots n} \varrho_i$$

If the aCFG is a correct representation of a program, then Definition 3.7 will provide corresponding values for reachability valuation and satisfyability valuation of the represented statement. When determining reachability valuation or satisfyability valuation of a hyper-node representing a decision, the external incoming and outgoing edges have to be used for summing up the input-valuation sets. The interpretation for $IV_R(x)$ and $IV_T(y), IV_F(y)$ given in Definition 3.7 and the Conservation-Axiom 3.2 both correspond with the statement of Corollary 2.1, because

$$\underbrace{\bigcup_{i=1\ldots m} \delta_i}_{IV_R(x)} = \bigcup_{i=1\ldots n} \varrho_i = \underbrace{\left(\bigcup_{i=1\ldots r} \varrho_i\right)}_{IV_T(y)} \cup \underbrace{\left(\bigcup_{i=r+1\ldots n} \varrho_i\right)}_{IV_F(y)}$$

3.3 Program Transformations

Several approaches have been made to formalize optimizing transformations for translation validation [34, 6]. Modelling program transformations is commonly based on some kind of rewriting rules that use a substitution to alter the program on relevant places [51]. In addition, an applicability condition is specified using some class of temporal logic [28], for instance, to specify the conditions under which a statement or a group of statements may be transformed [36].

The checks, whether or not the optimized version of an input program is equivalent to the original program, are mostly based on semantic verification conditions derived from data flow [61]. But they rarely take care about changes in the structure of a program that can alter execution paths. This focus on semantic aspects is also reflected in the logical construct of the applicability conditions, which is therefore of poor usefulness for code coverage analysis.

Graph transformations on control-flow graph representations [7] seem to be more reasonable for describing changes in the structure of a program. Graph transformations are described by graph grammars, which provide a mechanism to specify local transformations on graphs in a mathematical way [47]. The basic concept of a graph transformation mechanism is to search for certain occurrences of a specified pattern inside a host graph. Whenever such an occurrence is found, it is removed and a specified graph is added instead. Finally the added graph is connected

to the remainder of the host graph using some embedding rules. Dependent on the used rewriting strategy, the replacement can be focused on nodes or edges.

Graph transformations have been mentioned in this work as a possible useful approach to gain information about changes in the structure of a program and the involved changes in control flow essential for coverage-preservation analysis. The first idea was to keep track of changes in the input-valuation sets associated with the edges of an aCFG while transforming the aCFG in small replacement steps. Unfortunately it figured out, that this strategy failed, because it is not able to collect all the information needed to assess the corresponding change of input-valuation sets along the edges of the aCFG. If the reconstruction of the graph is done using a certain number of small steps, the intermediate steps of transforming a control-flow graph may produce incorrect graph structures or the produced intermediate aCFG is not semantically identical to the original program. As a consequence, essential information may get lost or may be corrupted while applying the transformation steps.

So instead of trying to keep track of the changes in the input-valuation sets associated with the control-flow edges during small transformation steps the program transformation is performed in one step and the statements of the transformed program are tagged with supplementary information to keep track of their origin in the original program. This idea is based on the observation, that a program transformation does not create the transformed program from sketch without considering the statements of the original program. It typically 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 statement in the original program or they are modified versions of a statement of the original program. Even newly created statements are not completely independent of the statements of the original program, since they are often added to correct the behavioural differences of statements modified during transformation.

Definition 3.8 *A node of the transformed program is said to* perform the same function *as a node of the original program, if the corresponding program statements produce identical results for the same set of input-data.*

A node of the transformed program is said to perform a similar function *as a node of the original program, if the corresponding program-statements produce different results for some input-data.*

A pair of nodes performing the same function or performing similar functions is called functional related nodes.

If two nodes, 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. For nodes representing simple statements, this behaviour is mandatory, because a node representing a simple statement has only one outgoing edge and therefore all input valuations entering the node are naturally mapped to the input-valuation set associated with the only one outgoing edge. For nodes and hyper-nodes representing conditions and decisions, the mapping of the incoming input-valuations 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 or decision performs the same function as in the original program.

On the contrary, if statements, especially 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. Since nodes representing simple statements and statement sequences have only one outgoing edge, changes of their functions cannot change the mapping of the input valuations from the incoming to the outgoing edges. So the assessment of changes in control flow caused by program transformations is focused on nodes representing conditions and hyper-nodes representing decisions.

The following definition will characterize the notion of functional-related nodes, established in Definition 3.8, and the two different levels of functional relationship more precisely:

Definition 3.9 *Let $P_1 \rightarrow P_2$ be a program transformation. 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, both representing a condition or a decision.*

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

$$IV_R(v) = IV_R(v') \implies IV_T(v) = IV_T(v') \quad \textit{and} \quad IV_F(v) = IV_F(v')$$

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

Corollary 3.2 *If v and v' are two functional-equivalent nodes, then the following additional relations must hold:*

$$\textit{(1)} \qquad IV_R(v) \subseteq IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \textit{and} \quad IV_F(v) \subseteq IV_F(v')$$
$$\textit{(2)} \qquad IV_R(v) \supseteq IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \quad \textit{and} \quad IV_F(v) \supseteq IV_F(v')$$

If two nodes v and v' are functional similar, the correlation between reachability-valuations and satisfyability-valuations is much more complex than for functional-equivalent nodes. If the reachability valuation $IV_R(v) = IV_R(v')$ is true, then one out of eight initial combinations for the satisfyability-valuations is possible for the behaviour of the node:

$$(1) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) = IV_T(v') \quad \text{and} \quad IV_F(v) \subseteq IV_F(v')$$
$$(2) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \text{and} \quad IV_F(v) = IV_F(v')$$
$$(3) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) = IV_T(v') \quad \text{and} \quad IV_F(v) \supseteq IV_F(v')$$
$$(4) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \quad \text{and} \quad IV_F(v) = IV_F(v')$$
$$(5) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \text{and} \quad IV_F(v) \supseteq IV_F(v')$$
$$(6) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \quad \text{and} \quad IV_F(v) \subseteq IV_F(v')$$
$$(7) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \subseteq IV_T(v') \quad \text{and} \quad IV_F(v) \subseteq IV_F(v')$$
$$(8) \qquad IV_R(v) = IV_R(v') \implies IV_T(v) \supseteq IV_T(v') \quad \text{and} \quad IV_F(v) \supseteq IV_F(v')$$

If the corresponding reachability valuation changes from equality to a non-strict superset or subset, then dependent on the combinations (1) to (8) the relations of the satisfyability valuations may stay unchanged, change from equality to a non-strict subset or non-strict superset or they may be undefined. The possible combinations for $IV_R(v) \subseteq IV_R(v')$ and $IV_R(v) \supseteq IV_R(v')$ in dependence of the initial combinations are listed in the table below:

	Satisfyability Relations							
	IV_T IV_F	IV_T IV_F	IV_T IV_F	IV_T IV_F	IV_T IV_F	IV_T IV_F	IV_T IV_F	IV_T IV_F
$IV_R(v) = IV_R(v')$	$= \subseteq$	$\subseteq =$	$= \supseteq$	$\supseteq =$	$\subseteq \supseteq$	$\supseteq \subseteq$	$\subseteq \subseteq$	$\supseteq \supseteq$
$IV_R(v) \subseteq IV_R(v')$	$\subseteq \subseteq$	$\subseteq \subseteq$	$\subseteq \times$	$\times \subseteq$	$\subseteq \times$	$\times \subseteq$	$\subseteq \subseteq$	$\times \times$
$IV_R(v) \supseteq IV_R(v')$	$\supseteq \times$	$\times \supseteq$	$\supseteq \supseteq$	$\supseteq \supseteq$	$\times \supseteq$	$\supseteq \times$	$\times \times$	$\supseteq \supseteq$

Each cell of the table contains the relation between the true-satisfyability valuations on the left-hand side and the relation of the false-satisfyability valuations on the right-hand side. The first row of the table defines the initial relation present in case of equality between the reachability valuations. The rows below show the combinations of the satisfyability-valuation relations if the equality condition for the reachability valuations is dropped. If the relation between particular satisfyability valuations cannot be determined, the corresponding position in the table is marked with $\times$.

For example, if in case of $IV_R(v) = IV_R(v')$ the satisfyability-valuations are related $IV_T(v) = IV_T(v')$ and $IV_F(v) \subseteq IV_F(v')$ then the first column shows the following further relations in the corresponding lines:

(1) $IV_R(v) \subseteq IV_R(v') \implies IV_T(v) \subseteq IV_T(v')$ and $IV_F(v) \subseteq IV_F(v')$

(2) $IV_R(v) \supseteq IV_R(v') \implies IV_T(v) \supseteq IV_T(v')$

If $IV_R(v) \supseteq IV_R(v')$ is true no relation exists 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 $IV_F(v')$.

In the aCFG representing the transformed program, related tagged nodes introduced with Definition 3.1(2) are used to create the link between a pair of functional related nodes. The related-node identifier of a related-tagged node is a pointer from a node of the transformed program to its counterpart in the original program.

Be aware, that the functional relations between nodes 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. But the program statements itself performing this task may be different. Consider for example a transformation as shown in Figure 3.4. 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

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

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.

3.4 Transformation Relations

Transformation relations or inter-CFG relations are relations between the input-valuation sets of edges in two different aCFGs. They can be determined by assumptions about the behaviour of a program transformation, by functional relationships between nodes of two different aCFGs and by using transitivity of local relations or already known transformation-relations.

Axiom 3.3 *Let s be the entry node of an aCFG P_1 and s' be the entry node of an aCFG P_2, which is the result of a transformation $P_1 \rightarrow P_2$.*

If δ is the input-valuation set associated with the outgoing edge of s and ϱ the input-valuation set associated with the outgoing edge of s', then $\delta = \varrho$.

The background of Axiom 3.3 is the inherent assumption that a transformation does not change the remainder of a program. If the aCFG comprises a complete program the assumption is obvious. If the aCFG represents only a fragment of a program, then it is assumed that the execution flow of the unchanged remainder of the program and therefore the input-data of the interface of the fragment does not change when transforming only the fragment.

In other words, it is required, that the behaviour of the entry-node of an aCFG representing a program P_1 is idempotent to an arbitrary number of transformations $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_z$. Furthermore, Corollary 3.1 implies, that also the union of the input valuations associated with the incoming edges of the exit node must keep unchanged during transformation. But the transformation $P_1 \rightarrow P_2$ may add some new sequences of edges possibly terminating in the exit node or the changes during transformation may cause a shift of input-valuations from one edge to another. So in contrast to the entry node each input-valuation set of the incoming edges of the exit node may differ compared with the input-valuation sets before the transformation. For example, the Loop-Peeling transformation described in Section 5.4 increases the number of edges terminating in the exit node.

Relations that are determined by assumptions about the behaviour of a program transformation emerge directly from the transformations post-conditions or from the applicability condition. This kind of relations is not always systematically derivable from other properties of the transformation. In most cases they are created using some prior knowledge about the behaviour of a transformation. For example, think of a loop transformation. A common requirement for a loop transformation is, that the number of executions of the innermost loop body must not be changed. Therefore all input valuations, executing the body of the innermost loop before the transformation must also execute the body of the innermost loop after the transformation, and therefore the input-valuation set on the edge representing the execution of the innermost loop must be the same before and after the transformation.

Most of the valuation-relations can be derived systematically using the knowledge about functional-related nodes. To be able to apply the rules for functional-similar nodes described in Section 3.3 above, the aCFG must be processed in a top-to-bottom order, starting with the basic equivalence of the entry-nodes. If a related tagged-node of the transformed program provides a pointer to a node of the original program and if the relation of the input-valuation sets associated to the incoming edges of two nodes are known, the relations of the input-valuation sets associated to the outgoing edges can be determined. Then the procedure can progress iteratively to the tail-nodes of the outgoing edges and so on. If the chain is broken somewhere, because no relations can be found, then sometimes processing the graph in bottom to top order starting at the exit-node may help. Moreover, deriving relations from functionally similar decisions may also help to jump over the gap of a broken relation chain.

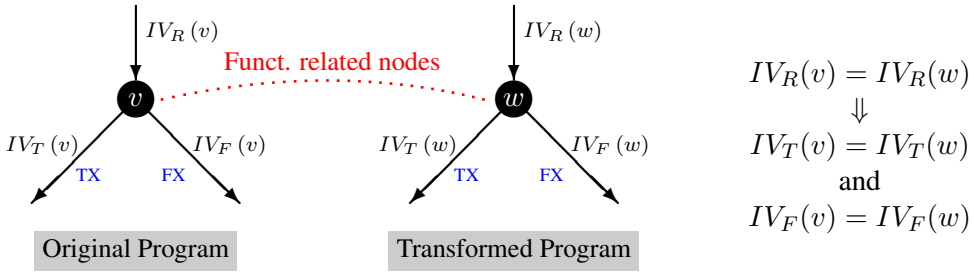


Figure 3.5: Determining inter-CFG relations by functional node relationships.

Transitivity is another chance to fill the gap, when the chain of subsequent inter-CFG relations is broken somewhere. Finding relationships by transitivity usually mean to use local relations inside one aCFG to close the sequence of relations. In addition, already known inter-CFG relations can be used to complete the chain.

3.5 Principle of Coverage Preservation Analysis

Basic Algorithm

The foundation, created in the first part of this chapter, will now be used to describe a principle algorithm for performing an analysis of the structural code coverage criteria described in Chapter 2. The analysis algorithm performs on simplified prototype versions of the investigated program fragments represented by aCFGs. The remainder of the program, not affected by the (optimizing) transformation, will be assumed to be concentrated inside the entry- and exit-node.

An aCFG P_1 of the program fragment where the optimizing code transformation under investigation takes place will serve as model for the analysis. A second aCFG P_2 will serve as model for the transformed version of P_1. The transformed aCFG P_2 will use related tagged nodes to setup functional relationships to the nodes of P_1. Beside these node relationships, initial input-valuation set relations between input-valuation sets of P_1 and P_2 can be used to describe, how the transformation affects the taxonomy of execution paths. After creating basic relationships of input-valuation sets, the code-coverage preservation criteria described in Chapter 2 will be used to compare the two versions of the program with respect to their ability of preserving a certain structural code coverage under investigation. In more detail, the principle algorithm looks as follows:

1. Create an aCFG P_1 for the program fragment under investigation.
2. Create a copy P_2 of the aCFG P_1 and perform the intended program transformations.
3. In the transformed program P_2 use related tagged nodes and setup pointers to pinpoint functional relationships between the nodes of P_2 and P_1.
4. Create new symbols for the input-valuation sets associated with the edges of the transformed aCFG to make the input-valuation sets of P_2 distinguishable from the input-valuation sets of P_1.
5. For each aCFG P_i $(i \in \{1, 2\})$ determine local relationships between the input-valuation sets:
   a) Setup the initial entry-node to exit-node relationship (Corollary 3.1).
   b) Walk through all nodes from $B(P_i) \cup C(P_i)$ and setup the relationships implied by the conservation-axiom (Axiom 3.2).
   c) Walk through all hyper-nodes from $D(P_i)$ and setup the relationships implied by the conservation-axiom (Axiom 3.2).
6. Create the basic inter-CFG relationships:
   a) Setup the initial relationship between the entry nodes and exit nodes of both aCFGs (Axiom 3.3).
   b) Add relationships derived from the characteristics of the performed program transformation.

c) Walk through the related-tagged condition nodes of P_2. If a known relationship between the incoming input-valuation sets of the node of P_2 and its functional-similar node in P_1 exists, register the relationship of the outgoing input-valuation sets.

7. Apply the preservation condition of the structural code coverage that should be investigated using a suitable procedure.

The concrete procedure for applying a certain preservation criteria depends on the structure of the used preservation condition. In most cases the procedure will walk through all nodes of P_2, belonging to a certain type. For each such node it will try to find a node of the same type in P_1, that fulfils a certain condition, required by the preservation criteria. If a corresponding node in P_1 can be found for all investigated nodes in P_2, then the preservation condition is proven for this special transformation-configuration.

A weakness of the algorithm with respect to an automatic detection of relationships is that the ability to find inter-CFG relationships may depend on the order the nodes of P_2 are investigated. Practical experience has shown that a top to bottom order, where the entry node is the top and the exit node is the bottom, works best. However, loop structures are always hard to handle, because the feedback links of the cycle are causing a stalemate. The feedback link is unknown as long as the start of the cycle is not analyzed, but the start of the cycle cannot be analyzed as long as the feedback link is not.

Example 1: Useless Code Elimination

Useless code elimination, sometimes also called dead code elimination, is an optimization that removes a statement that computes a value that is never used on any executable path leading from the position of the statement [40]. A common example for useless code is the assignment of a value to a variable that is never used again. A more detailed discussion of this optimization will be done in Section 5.2, and a code example is given in Figure 5.1.

Because of its simplicity this exercise is well suited to take a deeper look into the transformation steps. Figure 3.6 depicts the steps taken to convert the aCFG of a program fragment to an aCFG of the optimized program fragment. The nodes of the graphs are identified by integer-numbers drawn in the middle of the node-symbols. Entry node and exit node are always marked with an extra circle around the node. It is assumed, that node 3 of the original program is useless code, that should be removed.

Figure 3.6(a) shows the linear sequence of statements in its original structure. The input-valuation sets associated with the edges are denoted $\delta_1, \ldots, \delta_4$. The first step of the transformation (b) creates a one-by-one copy of the original graph. The copy uses related tagged nodes to keep track of the relations between the nodes of the original program and the transformed program. In the figure this is drawn with mapping symbols pointing to the identifier of the related node. To make the nodes of the copy distinguishable from the original graph, the identifiers of the copied nodes are replaced by numbers ranging from 11 to 15. The input-valuation sets are dropped in this step, because they must be recalculated after the transformation. Now the actual optimization step is performed (c). Node 13, which is assumed to be useless code, is removed and the edge originating at node 12 is reconnected with node 14. Finally, the edges are asso-

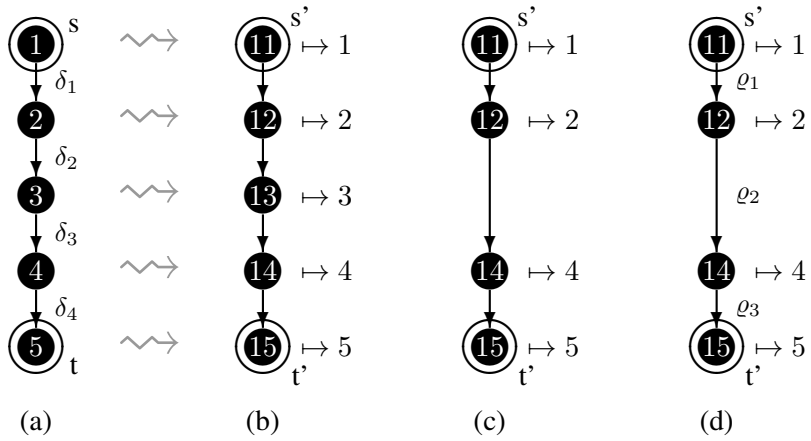


Figure 3.6: Transformation steps for useless code elimination.

ciated with input-valuation sets (d). To distinguish them from the input-valuation sets δ_i of the original program, different symbols ϱ_i are used.

Note, that following aCFG drawings will skip intermediate steps like step (b) and (c) in figure 3.6. The aCFG representing the original program and the aCFG of the final transformation result (the transformed program) will be drawn only.

As further preparation steps for coverage-preservation analysis, the local relations of the relevant graphs (a) and (d) have to be determined. This is very easy, since there are only simple statements involved. Walking through nodes 2, 3 and 4 of the original program (a) and the nodes 12 and 14 of the optimized program (d) reveals:

$$\underbrace{\delta_1 = \delta_2,\ \delta_2 = \delta_3,\ \delta_3 = \delta_4}_{\text{(a)}} \qquad \text{and} \qquad \underbrace{\varrho_1 = \varrho_2,\ \varrho_2 = \varrho_3}_{\text{(d)}} \tag{3.3}$$

The transformation relations are also easy to determine in this case, because $\delta_1 = \varrho_1$ must be true by default (Axiom 3.3). Transitivity then implies the equality $\delta_i = \varrho_j$ for all $1 \leq i \leq 4$ and $1 \leq j \leq 3$.

Based on these relations, preservation of statement-coverage can now be proved, using the condition from Theorem 2.1:

$$\forall b' \in B(P_2)\ \exists b \in B(P_1) \text{ with } IV_R\left(b'\right) \supseteq IV_R\left(b\right) \tag{3.4}$$

Because of Definition 3.7, the following equalities are evident:

$$IV_R\left(12\right) = \varrho_1,\ IV_R\left(14\right) = \varrho_2,\ IV_R\left(2\right) = \delta_1,\ IV_R\left(3\right) = \delta_2,\ IV_R\left(4\right) = \delta_3 \tag{3.5}$$

To perform the formal proof of (3.4), the preservation condition has to be shown for all nodes $b' \in \{12, 14\}$ in relation to the statement-nodes $b \in \{2, 3, 4\}$. Putting together (3.5) with (3.3),

the validity of the statement-coverage preservation condition $IV_R(b') \supseteq IV_R(b)$ is obviously true for all possible combinations of nodes b' and b.

Since the program fragment contains no conditions or decisions, the preservation conditions for condition coverage and decision coverage are *true* by default. This interpretation conforms to the fact, that CC and DC of the remainder of the program are not affected by that optimization. But be aware, that in this case the implication "decision coverage"$\Longrightarrow$"statement coverage" is inadmissible, since the part of the program relevant for the optimization contains no decision [41].

Example 2: Decision Distribution

The following example is a problem that could arise if the actual object-code implementation of a higher-level program construct is not known by the user or not exactly specified by the compiler-manufacturer. The programmer, for whatever reason, assumes that the two conditions of a branch-statement are always executed independent of the result of each condition. Unfortunately, the compiler translation of the branch results in two single decisions evaluating the branch condition with a shortcut-semantics. Figure 3.7 presents a possible example of the decision-distribution problem written in a high-level pseudo code language.

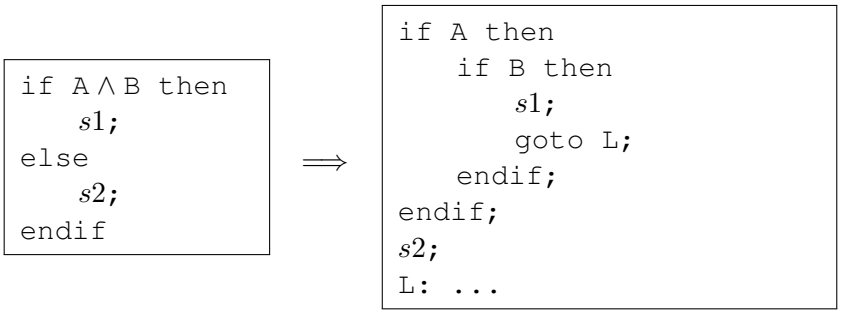


Figure 3.7: High-level pseudo code version of the decision distribution problem.

Figure 3.8 presents the aCFG representing the untransformed high-level program P_1 on the left and the final result of the object code level transformation P_2 on the right. In addition, the node equivalences relevant for the transformation analysis are shown next to the aCFG representing P_2. The transformation has not changed the semantics of any statement. Only how statements are linked together has changed.

Moreover, a characteristic property of the shown transformation is, that the statement sequences representing the *then* branch and the *else* branch in both versions of the program must be executed for the same input values. Therefore, the following transformation-relations must hold:

$$\delta_4 = \varrho_4 \quad \text{and} \quad \delta_5 \cup \delta_6 = \varrho_3 \cup \varrho_5 \tag{3.6}$$

Inspecting the local relationships of both aCFGs by walking through each node in a top-

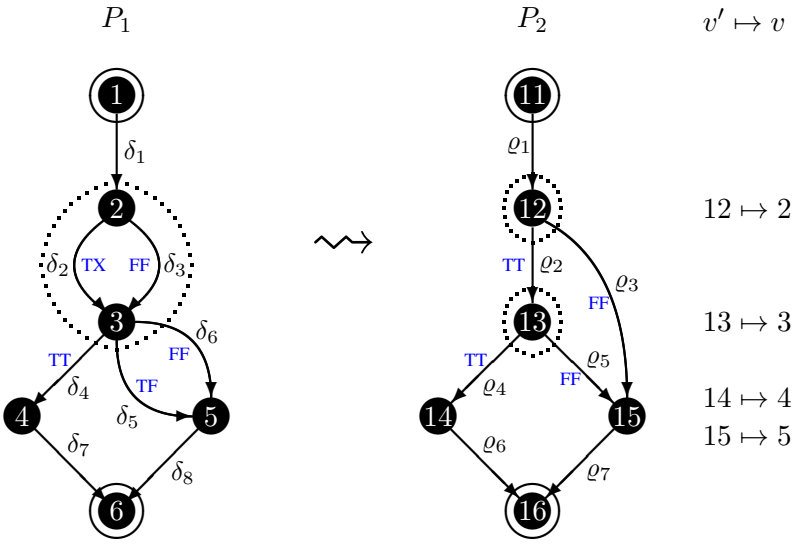


Figure 3.8: Two aCFGs representing P_1 and P_2 of the decision distribution problem.

down order, results in the following basic relations:

Entry/Exit: P_1	$\delta_1 = \delta_7 \cup \delta_8,\ \delta_1 \supseteq \delta_7,\ \delta_1 \supseteq \delta_8$
Node 2:	$\delta_1 = \delta_2 \cup \delta_3,\ \delta_1 \supseteq \delta_2,\ \delta_1 \supseteq \delta_3$
Node 3:	$\delta_2 \cup \delta_3 = \delta_4 \cup \delta_5 \cup \delta_6,\ \delta_2 \cup \delta_3 \supseteq \delta_4,\ \delta_2 \cup \delta_3 \supseteq \delta_5,$ $\delta_2 \cup \delta_3 \supseteq \delta_6,\ \delta_4 \cup \delta_5 \cup \delta_6 \supseteq \delta_2,\ \delta_4 \cup \delta_5 \cup \delta_6 \supseteq \delta_3$
Node $\langle 2,3 \rangle$:	$\delta_1 = \delta_4 \cup \delta_5 \cup \delta_6,\ \delta_1 \supseteq \delta_4,\ \delta_1 \supseteq \delta_5,\ \delta_1 \supseteq \delta_6$
Node 4:	$\delta_4 = \delta_7$
Node 5:	$\delta_5 \cup \delta_6 = \delta_8,\ \delta_8 \supseteq \delta_5,\ \delta_8 \supseteq \delta_6$
Entry/Exit: P_2	$\varrho_1 = \varrho_6 \cup \varrho_7,\ \varrho_1 \supseteq \varrho_6,\ \varrho_1 \supseteq \varrho_7$
Node 12:	$\varrho_1 = \varrho_2 \cup \varrho_3,\ \varrho_1 \supseteq \varrho_2,\ \varrho_1 \supseteq \varrho_3$
Node 13:	$\varrho_2 = \varrho_4 \cup \varrho_5,\ \varrho_2 \supseteq \varrho_4,\ \varrho_2 \supseteq \varrho_5$
Node 14:	$\varrho_4 = \varrho_6$
Node 15:	$\varrho_3 \cup \varrho_5 = \varrho_7,\ \varrho_7 \supseteq \varrho_3,\ \varrho_7 \supseteq \varrho_5$

Inspecting the condition-nodes of P_2 and their functional-similar nodes in P_1 reveals the following transformation-relations:

$$\text{Entry/Entry:} \quad \delta_1 = \varrho_1 \tag{3.7}$$

$$12 \mapsto 2: \quad \delta_1 = \varrho_1 \Longrightarrow \delta_2 = \varrho_2,\ \delta_3 = \varrho_3 \tag{3.8}$$

$$13 \mapsto 3: \quad \varrho_2 \subseteq \delta_2 \cup \delta_3 \Longrightarrow \varrho_4 \subseteq \delta_4,\ \varrho_5 \subseteq \delta_5 \cup \delta_6 \tag{3.9}$$

Note, that the transformation-relations (3.6) implied by the characteristic properties of the transformation are not completely derivable from functional relationships. The relationship $13 \mapsto 3$ for instance, only induces the relation $\varrho_4 \subseteq \delta_4$ and not equality as required by the transformation properties. Note also, that functional relationships for non-condition nodes are normally not mentioned explicitly. As considered before (Section 3.3) their one-by-one input-output relationships cannot directly cause a redirection of execution paths.

As an exercise, determination of local relations and transformation relations was done very extensively here. But for practical purposes it is not mandatory to derive all existing relations, because analysis usually only needs a small subset of them. To check preservation of statement coverage for example, it is sufficient to have the transformation relations from the characteristic properties of the transformation.

Preservation of Statement Coverage. The proof for preservation of statement coverage needs only the characteristic relations from (3.6). These relations immediately imply

$$\begin{aligned} \underline{IV_R\,(14)} &= \underbrace{\varrho_4 = \delta_4}_{(3.6)} &&= \underline{IV_R\,(4)} \\ \underline{IV_R\,(15)} &= \underbrace{\delta_5 \cup \delta_6 = \varrho_3 \cup \varrho_5}_{(3.6)} &&= \underline{IV_R\,(5)} \end{aligned}$$

and therefore statement-coverage is preserved.

Preservation of Decision Coverage. Expanding the predicate $touches\_ID$, the complete preservation condition for decision coverage looks as follows:

$$\begin{aligned} \forall d' \in D(P_2) \qquad & \exists\, d \in D(P_1) \;:\; (IV_T\,(d) \subseteq IV_T\left(d'\right)) \vee (IV_F\,(d) \subseteq IV_T\left(d'\right)) \\ \text{and} \qquad & \exists\, d \in D(P_1) \;:\; (IV_T\,(d) \subseteq IV_F\left(d'\right)) \vee (IV_F\,(d) \subseteq IV_F\left(d'\right)) \end{aligned}$$

The transformed program P_2 consists of two decisions: $\langle 12\rangle$ and $\langle 13\rangle$. These have to be correlated to the only one decision $\langle 2,3\rangle$ of P_1 to check the preservation condition.

$$\begin{aligned} IV_T\,(\langle 12\rangle) &= \underbrace{\varrho_2 = \delta_2}_{(3.8)} = IV_T\,(\langle 2,3\rangle) \implies true \\ IV_F\,(\langle 12\rangle) &= \underbrace{\varrho_3 = \delta_3}_{(3.8)} = IV_F\,(\langle 2,3\rangle) \implies true \\ IV_T\,(\langle 13\rangle) &= \underbrace{\varrho_4 = \delta_4}_{(3.6)} = IV_T\,(\langle 2,3\rangle) \implies true \\ IV_F\,(\langle 13\rangle) &= \underbrace{\varrho_5 \subseteq \delta_5 \cup \delta_6}_{(3.9)} = IV_F\,(\langle 2,3\rangle) \implies false \end{aligned}$$

Checking the preservation criteria for $IV_F\,(\langle 13\rangle)$ fails, because the distribution of the original decision $\langle 12,13\rangle$ redirects some paths to the edge $\langle 12,15\rangle$, and they are now missing at the

incoming edge of decision $\langle 13 \rangle$. As a consequence the missing valuations reduce the input-valuation set at the *false*-outcome of $\langle 13 \rangle$, and the required non-strict superset relation is not fulfiled. Therefore, the transformation does not preserve decision coverage.

Preservation of Condition Coverage. Expanding the predicate $touches\_ID$ results in the following preservation criteria for condition coverage:

$$\begin{aligned} \forall c' \in C(P_2) \qquad & \exists\, c \in C(P_1) \;:\; (IV_T(c) \subseteq IV_T(c')) \vee (IV_F(c) \subseteq IV_T(c')) \\ \text{and} \qquad & \exists\, c \in C(P_1) \;:\; (IV_T(c) \subseteq IV_F(c')) \vee (IV_F(c) \subseteq IV_F(c')) \end{aligned}$$

The transformed program P_2 consists of two conditions: 12 and 13. Possible candidates in P_1 for applying the preservation criterion are the conditions 2 and 3.

$$\begin{aligned} IV_T(12) &= \underbrace{\varrho_2 = \delta_2}_{(3.8)} = IV_T(2) \implies true \\ IV_F(12) &= \underbrace{\varrho_3 = \delta_3}_{(3.8)} = IV_F(2) \implies true \\ IV_T(13) &= \underbrace{\varrho_4 = \delta_4}_{(3.6)} = IV_T(3) \implies true \\ IV_F(13) &= \underbrace{\varrho_5 \subseteq \delta_5 \cup \delta_6 \supseteq \delta_5}_{(3.9)} = IV_F(3) \implies false \end{aligned}$$

The transformation does not preserve condition coverage, because the preservation condition for $IV_F(13)$ fails. Intuitively this result is expected, because the shortcut edge $\langle 12, 15 \rangle$ redirects some execution paths and therefore the *false*-satisfyability valuation of condition 13 is reduced.

3.6 Summary of Chapter 3

A formal control-flow graph based model for programs and program transformations is introduced. Based on this model, some basic rules to derive formal relations for coverage-preservation analysis are described. Examples are given to illustrate the usage of the model.

A basic procedure is described, how to utilize the control-flow graph model for coverage-preservation analysis.

CHAPTER 4

Automatic Analysis

This chapter describes, how the aCFG-Model and its application for coverage-preservation analysis are transposed to the mathematical software-system *Mathematica*. It combines the analysis approach described in Chapter 3 with the preservation conditions for structural code-coverage described in Chapter 2 to form a mathematical-computing system for investigating certain use cases with respect to code-coverage preservation.

Section 4.1 starts with the description of the internal implementation of the aCFG structure introduced in Section 3.1. The basic elements, used to compose nodes and edges are described first. The second part of this section describes how the nodes and edges and the different sets defining an aCFG are implemented and how they are linked together. The implementation description continues in Section 4.2 with some general graph function. Section 4.3 adds some remarks, how paths are handled in the *Mathematica* implementation. The processing steps for determining input-valuation-set relations are the focus of Section 4.4 and finally Section 4.5 describes the implementation of the coverage-preservation proofs.

Supplementary information for this chapter can be found in Appendix B, where the relevant parts of the implementation are listed. In addition, Appendix A gives a quick survey on the used *Mathematica* features.

4.1 aCFG Definition and Graph Functions

Basic Elements

Tags. Strings of arbitrary length and contents are used as tags for nodes and decision hypernodes. There is no predefined tag set implemented, so all tags can be constructed free from arbitrary character sequences when defining the aCFG structure. The tag string has mainly documentary functions with one exception for decisions. There is no need to keep tags unambiguous.

Node Identifiers. Identifiers are taken from the set of positive integers and must be unambigu-

ous inside an aCFG. Although it would be possible to use the same identifier set in different aCFGs it is good practice to use different ranges of integers for the identifiers of different aCFGs. This ensures a better distinguish ability for the nodes of aCFGs representing the original program and the transformed program.

Condition/Decision Label. The condition/decision labels of edges are implemented as strings with a length of up to 3 characters. An empty string is used for edges without condition/decision label. If not empty, the label string has a minimum length of 2 characters and each position in the label string is associated with a certain kind of information:

- The first position of the string is reserved for the *condition label*. The character "T" is used for condition label *true* and character "F" is used for *false*.
- The second position of the condition/decision-label string is associated with the decision label. The characters "T", "F" and "X" are used for decision outcomes *true*, *false* and *undecided*.

  For tagging feasible paths inside complex decisions, additional characters, representing a decision outcome of *true* or *false*, can be defined in the global variable `DecTrueFalseSet`. These additional characters are the corresponding implementation for the arbitrary decision-path markers $\alpha_1, ..., \alpha_r$ mentioned in Definition 3.3. The global variable for additional decision-path markers contains two lists: one for characters associated with a *true* outcome and one for characters associated with a *false* outcome. If a character from one of these sets is used for an external outgoing edge of a decision, it is treated as *true* or *false* dependent on its membership in the *true* or *false* set.
- The *functional relation character*, if present, is placed on the third position of the condition/decision label. This part supports the automatic generation of transformation relations for the head node of an edge. A detailed description will follow starting with page 62.

input-valuation set. The set of input-valuations associated with each edge is represented using unassigned *Mathematica* variables. Every free *Mathematica* variable can be used for this purpose. In this thesis, the symbols δ and ϱ with an index are used in most cases to tag the input-valuation sets of the original and the transformed program.

Path Marker. Path markers support some path-related calculations. Each path marker represents a certain path, traversing the edge it is associated with. Path markers are implemented as unassigned *Mathematica* symbols consisting of some symbol together with an index, like p_3 or π_7, for instance.

Graph Structure

In the *Mathematica* implementation the aCFG of a program is a nested structure of several lists handled like tuples or sets dependent on which part of the aCFG they represent. In the following description the terms *set* and *tuple* will be used synonymous with the term *list* to emphasize,

whether a list is treated as set or as a tuple. Please be aware that the internal implementation of sets and tuples is always a *list*.

In contrast to the formal mathematical Definition 3.4, the *Mathematica* representation of an aCFG uses only four components to define the internal graph-structure. These four components are:

- A set of *statement nodes* corresponding to the set $B(P)$. This set comprises the non-condition nodes.
- A set that corresponds to the set $D(P)$ of the aCFG definition, holding the decisions of the aCFG.
- The corresponding set for $R(P)$ with the edges of the aCFG.
- Two single nodes, one representing the entry node s and one representing the exit node t. As described in Section 3.1, these two nodes must be neither in $B(P)$ nor part of any decision in $D(P)$, and so they must not be members in the corresponding implementations of these sets.

Note, that the node set corresponding to $B(P)$ is used a little bit tricky to implement an extended statement-coverage preservation check. Please refer to the description of the implementation of the statement-coverage-preservation proof in Section 4.5 for more details.

The set $C(P)$ mentioned in the formal definition of the aCFG is not implemented explicitly. Since every condition must be uniquely assigned to some decision the corresponding set for $C(P)$ is constructed "on the fly" from the decisions if necessary. This avoids duplicates of condition nodes and possible inconsistencies in the assignments of conditions to decisions when defining aCFGs in the *Mathematica* framework.

Lists on the innermost level of the graph structure define nodes, decisions and edges. They are treated in a tuple-like manner. There are three types of such lists serving as basic building blocks:

Nodes. A *node* is a tuple comprising 2 or 3 elements. The first entry is reserved for the node tag, the second element is the node identifier. The third element, if available, holds the related nodes identifier. There is no explicit distinction between *tagged nodes* and *related tagged nodes*, except their number of elements. Both node types can be used at the same time for different nodes inside one graph.

Examples: The list `{"then",3}` is a possible definition of a *tagged node*, and the list `{"then",13,3}` is a possible *related tagged node* definition.

The tag of a single node has only documentary functions. In addition, tags of different nodes do not need to be different.

Decisions. A *decision hyper-node* is implemented as a flat list of arbitrary length with at least two elements. The decisions tag always occupies the first position in the list. The following entries define the condition nodes, the decision is composed of. The condition nodes can be represented by tagged nodes as well as by related tagged nodes without restrictions in combining different types of nodes.

Apart from its documentary function the tag of a decision is also used to distinguish loop-decisions and conditional-branch decisions. This is done by examining the tag-string of a decision for the occurrence of a special loop-keyword, defined in the global list `loopDecisionKeywords`.

Examples: The list `{"decision1",{"cond1*",11,1},{"cond2*",12,2}}` defines a decision hyper-node of two conditions using related tagged nodes.
The list `{"while",{"loopcond",1}}` defines a decision comprising one, non-related condition node. If the keyword "while" is member of `loopDecisionKeywords` (as it initially is), the second decision is classified as loop-decision.

Edges. An *edge* is implemented as a list, comprising at least four elements. The first element always defines the node identifier of the head node where the edge originates while the second element contains the node identifier of the destining node. The third element holds the condition/decision-label string and the fourth element stores the symbol for the input-valuation set associated with that edge.

Examples: The list `{1,2,"TX",`δ_2`}` represents an edge originating at a condition node. The list `{2,1,"",`δ_3`}` implements an edge originating at a simple statement node.

For some kind of calculations, a set of path markers may be created and appended as fifth element to the list. A path marker set contains one marker for each path that traverses the current edge. A more detailed description will follow starting on page 59.

On the topmost level, the four lists, defining the graph-structure, are bundled in a tuple, to allow handling of an aCFG as a unit. To get the certain components of an aCFG structure `P`, there is a set of retrieval functions:

`gB[P]`, `gD[P]` and `gR[P]` extract the corresponding sets for $B(P)$, $D(P)$ and $R(P)$ "as is". That means, that all these sets are returned, as they where originally defined.

`gC[P]` returns the virtual set of all conditions corresponding to $C(P)$. The set returned includes all conditions part of any decision defined in the corresponding set-implementation of $D(P)$.

`gST[P]` returns a list comprising two elements with the entry node on the first and the exit node on the second position.

`gV[P]` returns a set, that is simply the correspondence to the union of $B(P) \cup C(P)$.

Finally, the function `eCFG[B,D,R,ST]` constructs the corresponding structure for an aCFG by taking the corresponding sets for $B(P)$, $D(P)$, $R(P)$ and $ST(P)$. This function is intended to decouple the internal program defined structure of the aCFG from the logical structure, and to present a kind of "counterpart" to the way, components are returned by the retrieval function.

Retrieve Elements of Node, Hyper-Node and Edges

There are several functions implemented to extract the elements of nodes, decision hyper-nodes and edges in various ways. Most of these functions are intended to decouple the internals like position or actual presence of elements from the outside logical view. Others are designed to perform additional services, like extracting the comprehensive information of elements.

The following explanations are intended to survey the set of available functions. The complete function-set with a detailed description can be found in the *Mathematica* notebook (Appendix B):

Node Retrieval Functions. Functions like `gNodeLabel[]` or `gNodeId[]` are pure retrieval functions. Since these elements of the node structure are mandatory, these retrieval functions presuppose their availability and provide no "plan B" if they are missing.

On the contrary, the function `gNodeRelFunc[]`, which returns the identifier of the functional equivalent node in a second aCFG, checks the availability of the element first. It returns -1 as default if the element is missing in the nodes list structure, otherwise it returns the current value.

Most of the advanced functions work with node identifiers rather than directly with complete node tuples. To get the node tuple associated with an identifier, the supplementary function `gSelectNodeById[V,iv]` searches a node set V for the occurrence of a node with identifier iv. The result of this function is a set with zero or one element (taking for granted, that the node set V of the aCFG is correct).

Edge Retrieval Functions. `gEdgeHead[]`, `gEdgeTail[]` or `gEdgeValuation[]` simply retrieve certain components of an edge structure, like the identifier for the head node and tail-node or the input-valuation set symbol associated with the edge. As described before, mandatory elements are not checked for availability.

The access to the condition/decision label needs a two-staged procedure. In a first step, `gEdgeLabel[]` returns the whole string representing the complete set of information. To access a certain part one of the extra functions `gCLabel`, `gDLabel` or `gRLabel` must be applied to the label in a second step to get a single condition, decision or functional equivalence label.

Since edge path markers are optional, their retrieval function provides an appropriate default value `{}` (the empty set), if it is missing in the structure of the edge given as argument.

Decision Retrieval Functions. Although a decision can be represented by a list of arbitrary length, there are only two retrieval functions: one for the decisions tag, and one returning a set of the condition nodes the decision is composed of. The `gDecisionNodes[]` function returns a set with the nodes of a condition in complete form with all node elements. Since most of the functions work with node identifiers only `gDecisionIdSet[]` returns only the identifier of the decisions node set. In addition, `gDecisionRelSet[]` extracts the identifiers of the functional-related nodes of a decision. Knowing a set of

node identifiers of a decision the decision the nodes are belonging to can be found with `gSelectDecisionById[]`.

4.2 Elementary Graph Functions

Examining the Graph Structure

Functions like `gInEdges[]` or `gLabOutEdges[]`, for instance, are designed to extract the edges incoming to or outgoing from a certain node. These functions are provided in two versions: one for hyper-nodes and one for single nodes.

- The function for single nodes takes a single node identifier as argument and refers to the edges directly incoming to or outgoing from the node.
- The functions for the hyper-node version have the same name and the same structure of the argument-list as the function for single nodes, but always end with `H`. They take a set of node identifiers as hyper-node argument. The nodes defining the hyper-node are treated as a unit and the terms *outgoing edge* and *incoming edge* are transposed to the external edges of the hyper-node accordingly.

There are certain criteria available to select the edges of interest:

- `gInEdges[]` and `gOutEdges[]` simply select all edges incoming to or outgoing from a node or hyper-node.
- `gInDegree[]` and `gOutDegree[]` are counting the number of all edges incoming to or outgoing from a node or hyper-node.
- `gLabOutEdges[]` returns all outgoing edges of a node or hyper-node associated with a certain condition/decision label.
- `gSLabOutEdges[]` is similar to *gLabOutEdges*, but it allows selecting the outgoing edges of a node or hyper-node according to a certain part of the condition/decision label. The interesting part of the label is specified with a selection function that is responsible for extracting the interesting piece of information. The functions `CLabel` and `DLabel` are usually used for this purpose.

  Note, that `gLabOutEdges` and `gSLabOutEdges` have no corresponding functions for incoming edges, since selecting the edges by condition or decision label is only needed on outgoing side.
- `gPredEdges[]` and `gSuccEdges[]` determine corresponding incoming or outgoing edges, which share a node with a given edge. The difference between the two functions is how the given edge is treated. `gPredEdges[R,e]` treats the given edge e as outgoing edge of a node $head(e)$ and searches R for all occurrences of edges e' that fulfill the condition $tail(e') = head(e)$. `gSuccEdges[R,e]` acts exactly the other way round, searching R for all occurrences of edges e' with $head(e') = tail(e)$. In both cases, the result is returned in a set of edges.

Supplementary Functions

This paragraph deals with two kinds of helper functions: Search order functions and graph drawing functions.

Search Order Functions. These are functions, to determine a structure related search order for the nodes of a graph. In some cases it is necessary or preferred to traverse a graph in a *top to bottom* order. Search order functions provide a list, where the nodes of a graph are sorted in some reasonable sequence, which represents their position relative to the start-node.

Two different kinds of search orders are provided. *Breadth first search* order places nodes adjacent to the current examined node first before moving on. In the *depth first search* order nodes are stored with increasing distances from the start node first before continuing with the next node on the same level [49]. To guarantee that all nodes are included in the result-sets, all search order functions finally add remaining nodes to the end of the list. This situation can arise from nodes that are not connected to the start node of the search.

Graph Drawing. This set of functions supports drawing of simple pictures of the investigated graphs, based on the library `DiscreteMath`GraphPlot`. In *Mathematica* 5.2 the ability of the graph-plot library is restricted. Therefore, the capability of the main drawing function `gDrawCfg[]` is restricted too.

The drawings of a graph can be supplemented with node identifier and node-labels. In addition, the direction pointer and the condition/decision label for each edge can be drawn. Unfortunately, appealing plots of multiple edges between nodes are not directly supported by this version of the library. The chosen less-then-ideal solution is to draw multiple edges as a single edge, and add to it all condition/decision labels of the involved edges separated by colons.

Two versions of the `gDrawCfg[]` function are provided. Both versions take a structure representing an aCFG as first argument. The extended version takes two additional arguments: one for selecting to draw node tags and one for selecting to draw identifier for nodes. The simple version of `gDrawCfg[]` just draws the identifier for each node, since tags are often very space consuming because of their extensive length.

4.3 Path Handling

Path Construction

Paths are involved in the automatic analysis in two ways: firstly, they can be used to enhance the automatic local-relation analysis and secondly, they are needed for path-coverage analysis. Since a path is defined as a sequence of edges (Definition 3.5), a list containing the edges of the walk in the correct order is chosen as representation.

Two edge sets, one for the origin and one for the destination, serve as input for the path construction. The edges specified as origin of the paths are allowed to be outgoing edges of distinct nodes and similar the destination set may consist of incoming edges of distinct nodes.

Clearly, the construction must respect the direction of the included edges. In general, two specified nodes will be connected by more than one path, since an aCFG is a structure where paths can fork and join in an arbitrary way. On the other hand, there is no guarantee, that there exists a walk, connecting the specified start-node and end-node. However, the construction algorithm is designed to drop hopeless edge configurations, which will never reach the specified end of the path to guarantee the termination of the algorithm.

The function `gPathSet[R, ss, ts]` is the key function to determine paths possibly starting at a specified origin and ending at a specified destination. The function takes the edge set `R` (the edge set of the aCFG) as its first argument, the origination edge set as second argument `ss`, and the destination edge set is given in the third argument `ts`. The edge sets for start and end may include an arbitrary but non-zero number of members. The result of the function is a set of paths or the empty set, if no path between the edges of the specified edge sets exists.

`gPathSet[]` constructs paths in back-to-front order, starting with the edges from the termination edge set. Each edge of the end-set is initially assumed to be the last edge of an individual preliminary path. Therefore, each path of the result set is guaranteed to finish with one edge of the end-set, if a path exists.

After setting up the initial set of paths for investigation the algorithm try's to expand each path torso iteratively in backward direction. This is done by prepending the incoming edge of the head of its first edge to the path. If the examined node, joins n incoming edges, the currently investigated path torso is duplicated $n - 1$ times, and each incoming edge is prepended to one of these instances. If loops are found during path construction, all paths iterating the loop more than once are dropped. The construction of a path is finished, if either an edge from the start-set, a node without incoming edges or the entry node of the aCFG is reached.

There is no guarantee that path construction finally connects to any edge in the start-set. Thus there is no guarantee that any edge from the start-set is included in some path of the result set. But the algorithm will terminate anyway, because it will iterate loop-cycles at most one time and therefore cannot be trapped inside a loop. So it will somehow reach any node without incoming edge or all path torsos are dropped.

Unified paths

The path coverage preservation proof uses a re-organized set of the aCFGs edge set to determine possible execution paths. This re-organization removes multiple edges between nodes, and avoids additional paths in the structure of an aCFG caused by data-flow dependencies. The double edges connecting nodes 2 and 3 in the aCFG P_1 of Figure 3.8 on page 43, for instance, are an example for such a data-flow dependence. From the point of view of the program execution both edges are part of the same execution path, but the aCFG model requires having two different edges for the outgoing *true* and *false* outcome of a condition. If such kind of multiple edges are removed the resulting path is called *unified path*.

If multiple edges are removed, their condition labels and decision labels are joined in a certain way. If condition labels of both outcomes (*true* and *false*) are associated with the edges in question, the resulting condition label is set to the character "V". Otherwise the resulting label is *true* or *false* according to the values in the set. If the decision labels are all members of the same outcome set the decision label is set to the matching outcome *true* or *false*. Otherwise or if

one of the multiple labels contains “X”, the resulting decision component of the unified label is set to “X”. Functional equivalence symbols are omitted. Note, that these unified labels are only used temporary inside the function for proving the preservation of (scoped) path coverage.

Additional Path Functions

`gShortestSubPathSet[p,gG]` is a function to cut out sub-paths from a path given with the function argument `p`. The function returns a set of non-overlapping sub-paths of `p`, and each sub-path meets all the conditions described below:

1. All edges of a sub-path in the result set originate at nodes from the same node set. The node set is taken from a list of possible node sets supplied with the function argument `gG`.

2. If two edges are member of the same sub-path, they do not share a common head node. In other words, each node from the node set is used at most once.

3. The sub-path meeting condition (1) and (2) above is maximal. That means, that a sub-path is expanded with a subsequent edge, as long as the head of the subsequent edge is an unused member in the same node set as the other edges of this sub-path.

Note, that sections of the path `p`, that do not meet the conditions above, will be dropped. Note also, that the edges of different sub-paths are allowed to originate at nodes from different node sets. But all edges in the same sub-path are required to have head nodes from the same node set.

The function `gShortestSubPathSet` is usually used to pick out the sections, where a path traverses a certain decision, and to drop the edges outside the decision. In this case the given node sets, supplied with the function argument `gG`, are the investigated decisions of a program.

4.4 Input-Valuation Relation Processing

Input-Valuation Set Handling

Input-valuation sets are represented by symbols associated with edges of an aCFG. In addition to these input-valuation sets directly gained from the edges of the aCFG, unions of input-valuation sets must be handled to obtain representations for joins of execution paths. These unions must be treated in a symbolic way, since the actual contents of input-valuation sets is unknown. For programming technical reasons the internal representation of a union of input-valuation sets is a set of input-valuation set symbols. Single input-valuation sets are represented by a set containing the single input-valuation set symbol as the only one element. One advantage of this internal representation is that the union of input-valuation sets is reduced to simply flatten the set containing all operands of the union.

Example: The notation $\{\delta_i\}$ represents the input-valuation set δ_i inside the automatic analysis, and the notation $\{\delta_i, \delta_j, \delta_k\}$ represents the union $\delta_i \cup \delta_j \cup \delta_k$ of input-valuation sets.

The input-valuation-relation analysis is based on an auxiliary graph 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 *superset-or-equal* relations

implied by the basic properties of input-valuation relations. In other words, if $v = \{\delta_i, \ldots\}$ and $w = \{\delta_j, \ldots\}$ are two vertices representing input-valuation sets then arc $\langle v, w\rangle$ in the input-valuation relation graph indicates that $v \supseteq w$ is true. Note, that according to basic set-theory [14] the existence of $\langle v, w\rangle$ and $\langle w, v\rangle$ implies equality between v and w.

The structure of the *input-valuation-relation graph* represents the direct input-valuation relations gathered from Axiom 3.2 and other directly derived relations. But it 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 superset relation the graph is corresponding to is transitive. But the idea behind 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.

In the practical implementation of the automatic coverage-preservation analysis it was more efficient to always determine all possible relatives to the given input-valuation set. This is, because most of the coverage-preservation proofs need to determine several relatives to one input-valuation set, and most of the needed information is created as a side effect of the search. The main function for determining such relations is `vRelNodes[]`. It uses a *breadth-first-search* strategy starting at a given input-valuation set v in the input-valuation relation graph. Its purpose is to determine all input-valuation sets stored in the input-valuation relation graph, which are in superset-or-equal relation to the given input-valuation set.

The check, whether a given input-valuation relation set δ_i is a superset of some other input-valuation relation set δ_j or not, can be more or less reduced to check, whether or not δ_j is included in the set of relatives of δ_i returned by `vRelNodes[R, {`δ_i`}]`. If a coverage preservation proof needs to check the same node of some aCFG with the same kind of reachability valuation or satisfyability valuation, then the set of relatives must be determined only once and can be reused to check it against the reachability valuation or satisfyability valuation of different instances of another node.

In most cases, determining if a given input-valuation set is a member in the set of relatives returned by `vRelNodes[]` is a sufficient check for a superset-or-equal relation of the involved input-valuation sets, especially when checking coverage preservation based on a finished input-valuation-relation graph. In some rare cases this simple procedure may fail. Particularly during the construction of a relation graph the presence of obvious relations may be sensitive to the order of the construction sequence. So when using the relations constructed so far, the necessary relation may not be available directly. But other relations may be already present in the graph, which can substitute the lack of relations. If it is necessary to check, for instance, if $\delta_i \supseteq \delta_j \cup \delta_k$ is true, then for the simple method a vertices for δ_i and one for $\delta_j \cup \delta_k$ must be present to apply the simple check. But if $\delta_j \cup \delta_k$ is missing the check fails. On the other hand, if the relations $\delta_i \supseteq \delta_j$ and $\delta_i \supseteq \delta_k$ are both already available then the relation above eventually can be verified successful using an enhanced check.

To understand the method for the enhanced check consider a union of n subsets $B := b_1 \cup b_2 \cup \ldots \cup b_n$, each of them being a subset of another set A. Then $B \subseteq A$ is true. Now consider a "sub-union" $D := b_{i_1} \cup \ldots \cup b_{i_m}$ with b_{i_k} $(1 \leq i_k \leq n)$ being some element of $\{b_1, \ldots, b_n\}$.

Then D must be a subset of B, and therefore subset of A. This consideration can now be applied in the following way to check some relations like $\delta_i \supseteq \delta_j \cup \delta_k$ when $\delta_j \cup \delta_k$ is not directly available:

1. Determine the relatives for δ_i using the function `vRelNodes[R,`$\{\delta_i\}$`]`.

2. Calculate the union of all relatives. This can be simply done by "flattening" the result set and convert it into a set representation.

   For example, assume the result of `vRelNodes[]` is $\{\{\delta_r, \delta_j\}, \{\delta_r, \delta_k\}\}$ which is the internal representation for $\delta_r \cup \delta_j$ and $\delta_r \cup \delta_k$. Flattening the result and converting it into a set gives $\{\delta_r, \delta_j, \delta_k\}$ which is the internal representation for $\delta_r \cup \delta_j \cup \delta_k$.

3. Take the intersection of the input-valuation relation representation of $\delta_j \cup \delta_k$ which is $\{\delta_j, \delta_k\}$ with the calculated union of the relatives of δ_i. If the result of the intersection is $\{\delta_j, \delta_k\}$ (or in other words, if $\{\delta_j, \delta_k\}$ is a subset or equal), then $\delta_i \supseteq \{\delta_j, \delta_k\}$ must be true. Otherwise, the relation is false.

The function `vSubsetOf[sc,v]` supports the enhanced method for the relation check. It firstly performs the intersection operation between the set of related input-valuation sets `sc` and the input-valuation set `v` of the investigated node of the aCFG. Finally it returns a Boolean value to indicate, if the result of the intersection is equal to the investigated set `v`. This intersection-and-compare solution was chosen, because *Mathematica* does not directly support super-set operations or sub-set operations. Note, that the function does not flattening and unifying the input set `sc`, because it is often called several times with the same set `sc`, and at each call the flattening and unifying operations would be performed again and again. So it is left to the caller to do these operations.

Reachability and Satisfyability Valuation

Determining the reachability valuation $IV_R(x)$ and the satisfyability valuations $IV_T(x)$ and $IV_F(x)$ for some node x is implemented in a two stage process. First, a subset of edges tagged with the relevant input-valuation set information is selected. In the second step the valuation information is extracted from the obtained edge subset.

`cRIE[R, iv]` returns a set of all incoming edges of a node with identifier iv or hyper-node with identifier-set iv of an aCFG with edge set R.

`cIVR[R, iv]` is the *Mathematica* implementation of $IV_R(iv)$. The function returns the union $\delta_{i_1} \cup \ldots \cup \delta_{i_k}$ of the input-valuation sets of all incoming edges in the internal set notation $\{\delta_{i_1}, \ldots, \delta_{i_k}\}$.

`cSOE[R, iv, l]` returns a set of all edges marked with condition/decision label l outgoing from a node with identifier iv or hyper-node with identifier-set iv of an aCFG with edge set R. The function automatically accesses the condition or decision label according to the kind of node (single node or hyper-node) iv represents.

cIVS[R, iv, l] is the *Mathematica* implementation of the satisfyability valuation $IV_l(iv)$. It returns the union $\delta_{i_1} \cup \ldots \cup \delta_{i_k}$ of the input-valuation sets of all outgoing edges tagged with the condition or decision label $l \in \{\text{“}T\text{“}, \text{“}F\text{“}\}$. The result uses the internal set notation $\{\delta_{i_1}, \ldots, \delta_{i_k}\}$.

cSOD[R, iv] is a helper function to count the number of different outcomes of a node with identifier iv or a hyper-node with identifier set iv in an aCFG with edge set R. Typically the result is 1 for nodes representing simple statements or statement sequences, and 2 for nodes representing conditions or hyper-nodes representing decisions.

Note, that all these functions automatically determine the kind of node (single node or hyper-node) given as argument. The appropriate versions of the underlying support functions like `gInEdges[]` and `gInEdgesH[]`, for instance, are selected automatically.

Local Relations Graph Construction, the Simple Approach

Constructing the *input-valuation-relation graph* is the most important preparation step for automatic code-coverage analysis. Two methods are provided which differ in how local input-valuation relations are determined. This section describes the simple one.

Determining local input-valuation relations in a simple way starts by initially setting up the input-valuation set equality of the entry-nodes output and the exit-nodes input. Then it just examines each node of the interesting aCFG excluding entry-node and exit-node in some order. In the current implementation the node set of the aCFG is examined in the order actually defined with the use-case implementation. So the order of examination depends on the present order of the elements and on how identifiers are associated to nodes. For each node, the basic axioms, namely Axiom 3.1 and 3.2, are applied to determine the input-valuation relations locally obvious to the node as described in Section 3.2.

The determination procedure for single nodes first obtains the input-valuation sets of the incoming and the outgoing side. For simplicity, only single input-valuation sets and the union of all valuations entering or exiting a node are mentioned.

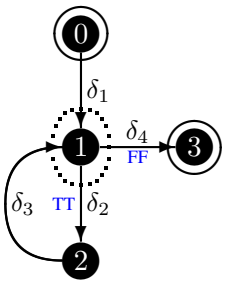
The helper function `vResolveValRel[inSet, outSet]` is used to construct pairs of related input-valuation sets. It combines `inSet` with each single element of `outSet`, and adds finally an equality pair for `inSet` with the complete `outSet`. Finally, the function `vResolveValRel[]` is called with swapped arguments, to reveal the complete result according to the conservation axiom. Each relation pair returned by `vResolveValRel[]` adds two vertices, each of them representing one of the involved input-valuation sets, and an arc connecting the vertices in the input-valuation relation graph. Since the edge list of the input-valuation relation graph is handled as a set, adding the same edge several times is an idempotent operation producing only a single copy of the edge. Note, that the input-valuation relation graph does not maintain an explicit set of vertices, because it is not needed for practical application.

The function `vResolveDecisionValuation[]` is similar to `vResolveValRel[]`, but it investigates input-valuation relations for a complete decision hyper-node and returns the determined relation pairs. The function `vValRelGraph[P]` is the main function for constructing a local input-valuation relation graph for an aCFG using the simple method of con-

struction. The function is called with a structure `P` representing the aCFG as argument. The function returns the whole bundle of relations found for `P`.

Local Relation Graph Construction, the Path Based Approach

During practical application of the automatic analysis with different use cases it has turned out, that the simple approach for constructing the local input-valuation-relation graph has some limitations. Especially for loop decisions it does not always obtain the results needed for a successful coverage-preservation analysis. Due to the focus of the simple algorithm on single nodes and single decision hyper-nodes, it has a restricted perception which limits the ability of the construction algorithm to identify global facts related to fork and joins of execution paths.




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

Simple input-valuation relations for node 1:

$$\delta_1 \cup \delta_3 = \delta_2 \cup \delta_4 \quad (4.1)$$
$$\delta_1 \subseteq \delta_2 \cup \delta_4 \quad (4.2)$$
$$\delta_3 \subseteq \delta_2 \cup \delta_4 \quad (4.3)$$
$$\delta_1 \cup \delta_3 \supseteq \delta_2 \quad (4.4)$$
$$\delta_1 \cup \delta_3 \supseteq \delta_4 \quad (4.5)$$

Some additional path-based relations for node 1:

$$\delta_3 = \delta_2 \quad (4.6)$$
$$\delta_1 \supseteq \delta_2 \quad (4.7)$$

Figure 4.1: Recognizing control-flow inside a simple loop configuration.

An example for the most common restriction of the simple approach is its inability to recognize edges going into and edges coming out from a loop. In the aCFG, shown in Figure 4.1, node 1 is a loop decision with a single condition. Edge $\langle 1, 2\rangle$ is the only edge going into the loop, and edge $\langle 2, 1\rangle$ is the only edge coming out of the loop. Since the loop has no intermediate points of exit, paths passing edge $\langle 1, 2\rangle$ must also pass edge $\langle 2, 1\rangle$, and furthermore the input-valuation sets δ_2 and δ_3 must comprise the same input-valuations. In contrast, the simple approach only will find the relations (4.1) to (4.5), listed on the right-hand side of Figure 4.1. All these relation are induced by using the conservation axiom for node 1. Looking at node 1 from a path-based point of view reveals, for example, the additional relations (4.6) and (4.7). The equality (4.6) is the corresponding formal proposition for the fact that all paths entering the loop also must leave it. In addition, the relation (4.7) formally expresses the fact, that some execution-paths will not enter the loop.

To avoid the limitations of the simple approach, an enhanced construction method called *path-based input-valuation graph construction* was developed for obtaining local input-valuation relations of an aCFG. The basic idea of the path based approach is to incorporate information

about execution paths through the aCFG into the edges before starting to evaluate the input-valuation set relations locally for each node. Analyzing the execution paths through the aCFG collects the path information used to generate path-marker sets, as described in Section 4.1, for all edges. The local evaluation can then benefit from this execution-path information, because it is able to identify interrelations caused by global control flow. So it is possible to generate specific relations between input-valuation sets associated with incoming and outgoing edges.

The path based valuation graph construction starts with determining possible execution paths through a given aCFG P and tags them with markers associated to each path. This job is done by the function `vPathTrace[]`, which first determines the set of execution paths. In a second step it uses the function `vAddPathMarker[]` to go through each path and to append a unique path marker ps_i to every edge of the aCFG which is in the examined path. The index i is incremented for every path so that each path marker represents a certain path, and i ranges therefore from 1 to the number of different paths. As a result `vPathTrace` returns a modified edge set $\overline{R}(P)$ of the aCFG where a set of path markers is appended to each element of $R(P)$.

Example: The following example demonstrates the modified edge set with added path markers corresponding to the aCFG illustrated on the left-hand side of Figure 4.1, and the call of the function `vPathTrace`, which is necessary to produce this result. The symbol `P` in the function call represents the aCFG structure implementing the aCFG in Figure 4.1.

```
MatrixForm[vPathTrace[gR[P],gST[P],Map[gDecisionIdSet,gD[P]],p]]
```

$$\begin{pmatrix} 0 & 1 & & \delta_1 & \{p_1, p_2\} \\ 1 & 2 & \text{TT} & \delta_2 & \{p_2\} \\ 2 & 1 & & \delta_3 & \{p_2\} \\ 1 & 3 & \text{FF} & \delta_4 & \{p_1, p_2\} \end{pmatrix}$$

The lines of the matrix show the modified edges $\overline{R}(P)$ of the aCFG with appended path-marker sets added for the different paths. In this case, two different execution paths are identified, and each of them is tagged with another path marker. Marker p_1 is used for the path not entering the loop, and therefore this marker is only used on the edge from the start node to the loop decision and then continuing from the loop decision to the termination node. Marker p_2 is used for the path that enters the loop and is therefore used for all edges of P.

Most of the work for constructing the input-valuation relation graph is done by the function `vPathRelValuations[R1_, R2_]`. The arguments `R1` and `R2` are two sets of edges the function should construct input-valuation relation pairs for. In most cases $R1$ contains the incoming edges and `R2` the outgoing edges of a node or decision hyper-node. Similar to the simple method of input-valuation relation graph construction relations are determined for the whole set `R1` with each element of `R2` and for all edges in `R2` with each single edge of `R1`. In addition, relations are determined for all pairs of edges from `R1` $\times$ `R2`.

The basic principle of the path based input-valuation relation construction is that the input-valuation sets are related in the same way as the path marker sets, which must be associated to each edge. This principle is derived from the assumption, that each path is triggered by a well-defined set of input-valuations. Therefore the input-valuations triggering a certain path must be present in the input-valuation sets associated with the edges traversed by this path. Supplemen-

tary to the conservation axiom additional relations are derived for each node based on the present path-markers associated with the incoming and outgoing edges. For the following description consider M_{in} being the set of path markers of some incoming edges of a node and M_{out} being the path-marker set of some outgoing edges of the same node. In the same way let V_{in} and V_{out} be the input-valuation sets associated to the incoming and outgoing edges mentioned with the path-marker sets. Then the logic for determining the additional input-valuation relations is as follows:

- If $M_{in} \supseteq M_{out}$, then the relation $\langle V_{in}, V_{out} \rangle$ is added to the input-valuation relation graph.
- If $M_{in} \subseteq M_{out}$, then the relation $\langle V_{out}, V_{in} \rangle$ is added to the input-valuation relation graph.

The described logic is implemented in the function `vResolvePathEquality[]` that is called several times as a sub-function of the input-valuation relation graph construction.

The results of the path-based approach for constructing the input-valuation relation graph have been compared experimental with the simple construction method. For loop-less structures both approaches produced the same results. For loop structures it turned out on one hand that for some use-cases the path based construction approach can acquire up to 40 percent more input-valuation relations than the simple method. On the other hand the price seems to be higher than the benefit. For more complex structures the number of paths grows rapidly with the complexity of the aCFG structure, even for small pieces of code. And on top of that, it turned out that even the additional relationships gained from the path-based approach are in some cases not sufficient for coverage-preservation analysis, and that it will be necessary to setup some relations manually. In such cases the local relationships gained with the simple method are often sufficient together with a few manually created one to successfully perform code-coverage preservation analysis. Unfortunately there is no absolute criterion to decide in advance, which method is more beneficial for the examined use-case. So it is decided empirical from case to case if the simple method or the path-based method is used to automatically determine the input-valuation relations. Nevertheless, in any case there is the possibility to supplement missing input-valuation relations manually.

Transformation Relation Graph Construction

Adding transformation relations is the second step in constructing the input-valuation relation graph. However, this part needs the local input-valuation relations of both aCFGs as a basement. Therefore the main function for constructing the transformation relation graph, the function `vTransRelGraph[uP, tP, hints, pathBased]`, calls the function for constructing a local-valuation graph first. In more detail, it calls the simple or path-based construction function exactly two times: Firstly for the original program `uP` and secondly for the transformed program `tP`. Both input-valuation relations are written into the same input-valuation relation graph. The caller determines with the Boolean function argument `pathBased` whether the function performs a simple construction or a path-based construction of the local input-valuation relations. The default value of the argument `pathBased` is `False`.

As described in Section 3.4, there are three relevant sources for gaining relationships between the input-valuation sets of the original and the transformed program. These input-valuation

sets are added to the existing local input-valuation relation graph produced in the first step. The relations are added in the following order:

1. Axiom 3.3 implies the initial equality between the output of the entry nodes of the two involved aCFGs. This initial equality is created first to establish an initial connection between the two aCFGs.

2. Relations derived from the properties of the program transformation. These are manually created input-valuation relations defined once together with the definition of the aCFG structures in the use-case implementation. They can include all kind of input-valuation-set relations, local relations inside each of the involved aCFGs as well as inter-CFG relations between the input-valuation sets of both aCFGs.

   The argument `hints` of the function `vTransRelGraph[]` takes manually created input-valuation relations. The input-valuation relations in the argument `hints` are simply added to the input-valuation-relation graph as defined without further processing.

3. Relations induced by functional relations between nodes of the transformed program and its functional-related nodes in the original program. This process will be described in more details below.

Note, that since the creation process also removes some relations already in the intermediate input-valuation relation graph the relation of the argument *hints* are added a second time at the end of the creation process. This makes sure, that they are added anyway.

Functional relationships between the nodes of the transformed program and nodes of the original program are the source of information for automatically detecting transformation relations. Definition 3.9 introduced the notion of functional-equivalent nodes and functional-similar nodes to describe the properties of functional-related nodes. But knowing whether a pair of related nodes is functional equivalent or functional similar is of less importance for automatic transformation-relation detection than knowing at which particular outgoing edge in the transformed program the input-valuation sets have changed compared with the original program. Therefore the implementation of the coverage-preservation-analysis framework focuses on outgoing edges of functional-related nodes of the transformed aCFG.

To describe the changes in the input-valuation sets, special symbols are associated with the outgoing edges of a node of the transformed program. These symbols are used as hints during the automatic determination procedure for transformation relations, to identify the particular edges where the input-valuation sets have changed compared with the corresponding outgoing edge of the related node in the original program. For convenience the functional relationship-symbol was incorporated as last part of the condition/decision label. This avoids the need to introduce a new component in the edge definition. The functional relationship is defined by one character. This part is called the *relation label*, and it can be accessed with the function `gRLabel` (ref to Section 4.1 for more details). At the moment, the following symbols are supported to define functional relations between nodes (please refer also to the description in Section 3.3):

= for equal output. If the node receives the same input valuations on its incoming edges, the input-valuation set associated to the edge marked with “=” will be the same as in the functional-related node in the non transformed graph.

Note, that this is the default value. If no functional-relationship symbol is assigned to an edge, the symbol is always assumed to be “=”.

+ for *amplified* output. This symbol means, that if the incoming input-valuations of the head node of the edge are the same as in the non-transformed program, the valuation on the edge will be a superset of the comparable edge in the non-transformed program.

- is the same as "+" but in the other direction. If the input of functional-related nodes is equal, the output of the node in the transformed program on this edge will be less than the comparable edge in the original program.

X is a special implementation specific symbol used to suppress the creation of a transformation relation for the marked condition outcome of a node when creating the input-valuation relation graph. This functional relationship symbol is intended to avoid interferences between manually created transformation relations (defined as hint argument when calling the function for transformation relation graph creation), and automatically created relations. If “X” is present on a path in a decision, it overrides all other functional relations and no relation is generated for this outgoing edge of the decision.

Different to condition/decision labels there is no distinction between functional relationship of conditions and decisions. In case of a decision only the edges outgoing from the decision hyper-node are relevant. The functional relations along the path inside the decision are not mentioned. Like condition/decision labels the functional relationship is meant to be valid for the outgoing side of a node/hyper-node only.

Note, that if the functional relationship of all outgoing edges of a related node or related hyper-node in the transformed program is “=”, then the node is functional-equivalent to its counterpart in the original program. If a symbol different to “=” is associated with at least one outgoing edge, then the related nodes are functional similar.

If more than one edge is involved for the outcome of a condition or decision, then the functional relation must be determined as a logical combination of the relationship symbols of each involved edge. To do this, firstly all relation labels are collected and putted into a combination-set, removing all duplicates of the symbols. Then the following rules below are used to calculate the resulting common functional-relationship symbol of the combined edges:

1. If the combination set contains the symbol “X” (a single “X” or “X” together with any other symbol) or if it contains conflicting symbols (“+” and “-”), then the common result is “X”.

2. If the combination set contains only “=” then the common result is “=”.

3. If the combination set contains “=” with “+” or a single “+”, then the common result is “+”. If it contains “=” and “-” or a single “-”, then the common result is “-”.

The following table shows how the input-valuation set of an edge in the transformed program is effected dependent on the functional relationship associated to the edge and in relation to the proportion of the input-valuations of the incoming edges of the functional related nodes. The symbol "$\subseteq$" for the input of the transformed node in the table means that the reachability-valuations of the transformed node are a non-strict subset of the reachability-valuations of the functional-related node in the original program. "$\supseteq$" means, that the input-valuations of the transformed node are a non-strict superset compared with the reachability-valuations of the related node of the original program. The symbol "=" describes equality between the reachability-valuations of the functional-related nodes. The symbols on output side show, if the input-valuation set associated to the examined outgoing edge is equal, a non-strict subset or non-strict superset in the transformed program, compared with the corresponding edge in the non-transformed program. The character "$\times$" means, that the output cannot be qualified because of the unfavourable combination of the reachability-valuation relation and the kind of change defined by the relationship-symbol. Another reason for the "$\times$"-symbol can be, that the outgoing edge contains a suppression symbol "X".

			Equivalence symb. on outg. edge(s)					
			+	=	-	X		
Input of	$\rightarrow$	$\subseteq$	$\times$	$\subseteq$	$\subseteq$	$\times$	$\leftarrow$	Output of
transformed	$\rightarrow$	$=$	$\supseteq$	$=$	$\subseteq$	$\times$	$\leftarrow$	transformed
node	$\rightarrow$	$\supseteq$	$\supseteq$	$\supseteq$	$\times$	$\times$	$\leftarrow$	node

The function `vCondDecRelFunc` calculates the resulting relationship for a condition or a decision. The function automatically determines if the input is a node-index or a index-set representing nodes of a decision. In any cases only the outgoing edges labelled with a condition/decision label are taken into account. If there is more than one outgoing edge then the function returns a combined relation symbol that is calculated with the rules described above. Remember that for a decision only the outgoing edges and no internal edges are relevant to calculate the result.

4.5 Preservation Proofs

General Remarks

The functions for preservation proofs are all following the same scheme. Each proof function has the form `cXXPres[P1,P2,RG]`, where `XX` is the letter combination `SC` for *statement coverage*, `CC` for *condition coverage*, `DC` for *decision coverage*, `MCDC` for *modified condition/decision coverage* or `PC` for *(scoped) path coverage*. The argument `P1` supplies the function with the aCFG structure representing the original program while `P2` is the aCFG for the transformed version of *P1*. Finally, the argument `RG` holds the *input-valuation relation graph* containing all local relations and the transformation relations of the input-valuation sets in both programs. Since all proofs use the same input-valuation relation graph it was decided to construct it in the run-up of the preservation proof and hand it over to the functions as argument `RG`.

All proof functions return the Boolean value `True`, if the preservation condition has been verified successfully. Otherwise `False` is returned. That the proof function was not able to verify the preservation condition may point out that the examined transformation does not preserve the investigated code coverage. Missing relations between the involved input-valuation sets however may also cause a negative result. In other words a positive result is always accurate, because it is based on the existence of relations. A negative result can be created by the existence of "wrong" relations as well as by a lack of information.

In addition to the proof functions, most of the additional predicates used in the coverage preservation theorems 2.2, 2.3, 2.4 and 2.5 are implemented as well. Of course, most of their arguments are adapted to serve the needs of the *Mathematica* implementation. Instead of nodes representing some statements, for instance, often a list comprising the set of relational input-valuation sets is used as argument. This avoids unnecessary multiple calculations of input-valuation sets related to $IV_R(x)$, $IV_T(x)$ etc., if some node x is used frequently in successive calls of the same function.

All proofs can be executed manually by taking the necessary steps by hand. Examples for manually executed proofs can be found in the code coverage notebook. The code coverage notebook also provides some functions doing all steps automatically:

`cUseCaseAnalysis[name,p1,p2,hints,pathBased]` performs all necessary steps to execute all coverage-preservation proofs for one use case. The arguments of this function are:

name with a text string containing a name for the use case.

p1 holding the aCFG structure for the original program.

p2 holding the aCFG structure for the transformed version of *p1*.

hints a set of input-valuation relations characterizing the transformation. The caller can supply this set with any kind of input-valuation relations, independent whether they are local relations or transformation relations. But note, that these relations are taking effect for constructing transformation relations only, not for local relations.

pathBased is a Boolean value determining whether the local valuation graph construction should be done using the path based approach (if the argument is `True`) or the simple approach (if the argument is `False`).

`cPresAnalysis[outLevel]` is the "do everything" function, performing all implemented preservation proofs. It takes an integer value `outLevel` as the only argument. This parameter determines the grade of details of the progress output printed during execution. The function uses an internal map to translate the integer value to correlating assignments of the set variable `outputSet`. The map can be found at the beginning of the function definition in the code coverage notebook, where it can be modified and expanded if needed. However, a negative integer always means "totally silent" – no intermediate output is produced in this case.

The global variable `outputSet` is a set of integer numbers used to control the details of functions output prints. It determines the kind of information that should be printed while

processing preservation proof functions. The content of the variable is valid for nearly all kinds of functions, not only for high level functions. Each integer number used as element of `outputSet` represents a special level of information. The higher the value of the set element, the more detailed is the output. The correlated information is printed during processing, if the certain element is member of `outputSet`. An empty set produces no output. During manual evaluation the `outputSet` can be changed at any time by adding or removing values to or from `outputSet`.

The following description is a guideline, how the elements of `outputSet` can be used to produce/get certain kind of output:

- (0) ... high-level information produced on use-case level and above, e.g. printing which use-case is processed.
- (1) ... prints the input (the arguments) of use-cases.
- (2) ... prints information about steps done to process a certain preservation proof and information about preparation steps.
- (3) ... presenting important result of proof processing, e.g. printing final transformation relations.
- (4) ... prints information about internal steps when processing preservation proofs., e.g. print information about predicates used in preservation proofs.
- (5) ... informs about important sub-results, e.g. which nodes are related to each other.
- (6) ... more detailed output of internal results.
- (7) ... currently unused.
- (8) ... currently unused.
- (9) ... prints pure debugging information, e.g. print name and arguments of called sub-functions.

Be aware, that the usefulness of some kind of information is sometimes hard to classify and may depend on the point of view of the user. So take the list above rather as a guideline than as a strict rule.

Statement Coverage

The preservation proof for statement coverage in the function `cSCPres[]` is a straightforward implementation of the preservation condition (Theorem 2.1). The algorithm replaces the "for all" quantifier with a loop, running over all nodes of the set `gB[P2]`. For each node of $P2$ it scans the nodes of `gB[P1]`, whether or not they satisfy the required superset condition for the reachability valuations. All nodes of $P1$ satisfying the preservation condition with respect to the examined node of $P2$ fill a result set. The test for one node of P_2 is classified as successful,

if the result is not the empty set. Although it would be sufficient to abort as soon as a pair of nodes is successfully tested, the function performs a complete test comprising all nodes of $P1$. This behaviour has technical reasons as well as documentary reasons. So a total number of tests performed is equal to $|B(P2) \times B(P1)|$.

If the set B comprises only the simple statements and statement sequences of the program (as it is originally defined in Section 4.1), the statement coverage test is performed in a "classical" way without mentioning the conditions. But since the preservation proof function performs the statement-coverage preservation check with everything that is member of $B(P_2)$ and $B(P_1)$, the sets $B(P_i)$ can be used a little bit tricky to perform a enhanced check. If the sets hold also copies of the conditions of the programs, the conditions are treated like simple statement and tested if their reachability valuation meets the preservation condition. However, every possible valid assignment of the set $B(P_i)$ would be conceivable.

The analysis function `cPresAnalysis[]` uses this behaviour to perform both kind of statement coverage analysis. The classical analysis is performed with the original sets $B(P_2)$ and $B(P_1)$. For the enhanced statement-coverage preservation analysis the sets $B(P_2)$ and $B(P_1)$ are temporary changed to $B(P_2) \cup C(P_2)$ and $B(P_1) \cup C(P_1)$ to produce unified sets of simple statements and condition statements.

Condition and Decision Coverage

Since the preservation proofs for condition and decision coverage are very similar, they are described together in this section. The only difference of the proofs is the use of decision hyper-nodes instead of condition nodes when proofing preservation of decision coverage. As well as that, the implementations of both proofs have the same algorithmic structure, and they differ only in the use of condition nodes or decision hyper-nodes as objects of test. Both proofs are distributed on two functions according to the use of an additional predicate in theorem 2.2 and theorem 2.3 respectively.

`touchesID[]` implements the helper predicate from definition 2.1. The implementation is a one-to-one transposition of the predicate condition. The only difference is that it takes the set of input-valuation sets related to $IV_R(x)$ as argument, serving as a substitute for the node x. As the predicate in the definition the function works correctly independent of whether it is called with a single condition or with a decision.

`cCCPres[]` and `cDCPres[]` provide the algorithmic implementation of the formal preservation conditions for condition and decision coverage. The only difference between `cCCPres[]` and `cDCPres[]` is that the condition coverage check iterates over the nodes from `gC[P2]` and `gC[P1]` while the decision coverage check iterates over the hyper-nodes of `gD[P2]` and `gD[P1]`.

Like the function for statement coverage, they are implemented in a straightforward manner using the same principles. The nodes, relevant for testing the preservation condition are taken from the appropriate sets of the program representations $P2$ and $P1$. Each relevant node of $P2$ is tested in a loop with all relevant nodes of $P1$. The examination fills two local result sets, one for the true satisfyability valuation and one for the false satisfyability valuation. The local result sets contain all nodes of $P1$ meeting the preservation condition. The overall result is again calculated by testing these result sets for the empty set. As described for statement coverage, the

test is done completely, although is would be sufficient to abort as soon as a positive test result is detected.

Modified Condition/Decision Coverage

The preservation condition for modified condition/decision coverage (Theorem 2.4) differs from the other preservation conditions described so far in this section. It requires the existence of some pairs of subsets of the input data, which elements fulfil the MCDC conditions in a certain manner. But the only knowledge about the valuations inside the program representations comprises the input-valuation sets associated with the edges of the aCFG. So the first task must be to find some candidates for subsets of the input data, which can later be used for the MCDC preservation proof. This makes the automatic evaluation of the MCDC preservation proof a little bit wicked.

The function `constructIDcand[R, d, c]` returns a list of possible characterizations for ID-sets related to a condition c of a decision d in the program, defined by its execution relation R. The function applies a trick to find candidates for the subsets ID_1, ID_2 and ID_{tmp} mentioned in Theorem 2.4. It uses the formal conditions from Definition 2.7 in reverse, assuming that MCDC is fulfilled in P_1. As a result it returns a list of possible characterizations for ID-sets related to decision d and its condition c.

The characterization for such an ID-set is a 3-tuple $\langle \gamma, \delta, \sigma \rangle$. Here γ is the input-valuation set related to the condition c, determined by $IV_T(c)$ or $IV_F(c)$ in the auxiliary predicate *mult_control_expr*. The element δ is the input-valuation set related to the decision d determined by $IV_T(d)$ or $IV_F(d)$ in *mult_control_expr*. And finally σ is a set containing all input-valuation sets in all condition $c'' \neq c$ with an empty intersection $ID \cap IV_T(c'')$ or $ID \cap IV_F(c'')$, determined by the *isInvariantExpr* condition of *uniqueCause*.

The practical construction of this list is based on analysis of paths starting at the entry of the decision and terminating at the outgoing *true* edge and the outgoing *false* edge of the decision respectively. The edge sequence is constructed in a two-step process. In the first step the paths from the decisions entry up to the incoming edges of the currently investigated condition and from the outgoing *true* and *false* edges of the condition to the outgoing *true* and *false* edges of the decision are calculated separately. In the second step all feasible combinations of the partial paths obtained in the first step are constructed. The input-valuation sets of the edges along a feasible path-combination are then collected as possible characteristics for the input-valuation sets $ID1$ and $ID2$.

The proof function `cMCDCPres[]` for preservation of modified condition/decision coverage first of all calculates a list of possible candidates for the sets ID_1, ID_2, and ID_{tmp} using the function described above. Then the preservation condition is executed iteratively in two loops for all decisions and conditions of P_2 in correlation with the decisions and conditions of P_1. The list of possible candidates for ID_k is used in the predicates to check if they lead to a feasible path configuration in the examined decision. Since the sets ID_1 and ID_2 must be the same for all checks, the different predicates are called one after the other using a system of systematically excluding unsuitable candidates. The first predicate always starts with the full list of all ID_k-sets and returns a subset of candidates that fulfil the predicates condition. This reduced list is then handed over to the second predicate which removes again candidates not meeting the predicates condition, and so on. At the end a list of candidates fulfilling all conditions of all predicates is

left. If this list is not empty the check was successful.

Since the preservation proofs are only able to handle sets of valuations they never deal with single elements from the sets ID_k, but with the whole sets instead. This procedure is permissible, because the elements of the ID_k sets are always used in a way similar like the following excerpt from the coverage-preservation condition:

$$\forall \langle id_1, id_2 \rangle \in ID_1 \times ID_{tmp} \quad : \quad unique\_Cause(c, d, id_1, id_2)$$

For the same reason the predicate *unique_Cause* and all its sub-predicates are defined slightly different. Instead of using all possible combinations of pairs of elements and checking the condition $td_i \in IV_T(x)$ etc. it takes the whole set ID_i and checks $ID_i \subseteq IV_T(x)$ etc. If this check succeeds the original proposition of *unique_cause* must be *true*, because of the definition of the non-strict subset relation ($A \subseteq B :\Longleftrightarrow \forall a \in A \Rightarrow a \in B$).

As a consequence of modifying *unique_cause* (Definition 2.2) in the way described above, the first part of this predicate *control_expr* will become identical to *mult_control_expr*. Therefore the predicate *control_expr* is not implemented, and the predicate *mult_control_expr* replaces it in an appropriate way.

Using *mult_control_expr* for checking the first proposition of *unique_cause* makes the second use of the *mult_control_expr* check unnecessary when trying to proof the MCDC conditions, and so it is omitted.

(Scoped) Path Coverage

Programs can be divided into scopes in a very sophisticated way [60]. Such divisions can include overlapping segments as well as nested segments. Therefore the number of ways to structure a program into segments is very huge. To avoid all this complexity the implementation of the automatic preservation proof for scoped path coverage restricts the mentioned segmentations of the examined programs to those, which include the whole investigated fragment inside one segment. So the automatic preservation proof excludes the "for all scopes" clause from its area of responsibility. The scoped path coverage is reduced to a kind of "path coverage" problem.

The path coverage preservation proof function `cPCPres[]` only works on unified paths as described on page 54. Modifying the edge sets of both aCFGs to include only unified edges and use these when determining the paths produces the unified paths. The rest of the function is relatively straightforward to Theorem 2.5. The function `cCondPathTrace[]` is used to perform the required distribution of the conditions with respect to their outcome to different sets. This function is also responsible for handling the "in-official" condition outcome "V", created during the joining of labels of the unified edges. The predicate *is_CondTF_enclosed* is implemented accordingly. In addition, `allConditionsEnclosed[]` implements the multiple check for *is_CondTF_enclosed* of a complete set of conditions.

4.6 Graph Transformation Functions

The work done for this thesis also included some experimental implementations of graph transformation functions. The principle of these functions was inspired by [17]. The scope of this

work was to gather some information useful for automatically tracking the changes of the input-valuation sets caused by the program transformation. As described in Section 3.3 it turned out that this goal could not be reached with the taken approach based on input-valuation sets.

To get information about changes of the input-valuation sets it is necessary to transform the graph in small steps. If the steps are too big, the intermediate steps will reveal no information how the input-valuation sets are changed by the transformation. Two different procedures have been considered to investigate the transformation problem:

- Classical *graph rewriting* [47] uses a "cut-and-paste" strategy to transform a graph. Each rewriting rule removes some part of the host graph, associates a new sub graph to it and uses then some embedding rules to connect the added sub graph with the remainder of the host graph. More complicated rewritings can be realized applying a chain of graph rewritings, each of them modifying the intermediate result of the last rewriting.

  Transposed to the aCFG model this procedure means that the intermediate results are not valid representations of some program, especially they may not be even valid aCFGs. Valuation information on adjacent edges may get lost, because the operation may result in an unconnected graph. In general it is hard to assess how relations between input-valuation sets change.

- In contrast to the classical "cut-and-paste" strategy, a "first-insert-then-remove" strategy was considered. This method changes the aCFG in a way that new structures are inserted first before some structures are removed. This produces always a formally correct aCFG. Edges can be reconnected immediately, which makes it easy to keep the relevant valuation information.

  Unfortunately, this method is also insufficient, because the intermediate results are not semantically equivalent representations of the investigated program. Consider a transformation, for instance, which swaps the conditions of a branch statement. Applying the "first-insert-than-remove" strategy would mean that a new copy of one of the conditions is added before the not longer needed copy is removed. The intermediate result need not always be a semantically equivalent version of the program, and therefore relations between valuations may change in an unexpected manner.

Finally the experiments where stopped, since it is not the main focus of this thesis. As an essence of these experiments the *functional relationship* approach (see Section 3.3) has been developed and involved into the aCFG model. This approach allows to include facts about the transformation into the aCFGs and determine some transformation relations partially automatically.

4.7 Restrictions of the Current Implementation

This section provides some final notes concerning the limitations of the taken approach. Some of them are specific to the formal model design while others are caused by implementation decisions.

- *Loop structures.* In the current implementation, it is assumed, that loops have only one single entry point and only one exit point. Multiple exits to be like, for example, the `break` statement in C/C++ are not supported yet.

- *Support for coupled conditions.* This is a restriction of the aCFG model. No information is provided to identify coupled conditions inside a decision and moreover each condition is allowed to be included in a decision only once.

  This seems to be a minor disadvantage, since the classical modified condition/decision coverage definition the preservation condition is based on cannot be fulfilled with coupled conditions anyway.

- *Number of condition/decision outcomes.* In the current model the number of results evaluating a condition or decision is restricted to *true* and *false*. It should be possible to extend the model and the implementation for supporting more than two outcomes. This would require changing some of the theoretical background, for example, satisfyability valuation and preservation conditions as considered in Section 2.3. Most of the involved functions of the framework would need a change, especially functions related to input-valuation handling described in Section 4.4. In addition, it would be necessary to adapt the proof functions, described in Section 4.5, to involve the changed coverage-preservation checks.

- *Multiple outgoing edges from the entry node.* At the moment, the definition of the aCFG structure (ref. to Section 3.2) only allows one edge outgoing from the start node. Multiple outgoing edges from the entry node would be a possible enhancement to model optimizations executing parts of the program in parallel. Multiple entry points to the investigated program fragment would be another possible application of multiple outgoing edges of the start node.

- *Hidden decisions and conditions.* The model does not address explicitly the possibility of nested conditions or decisions. These situations can arise if conditions comprise function calls with further decisions inside. This thesis assumes, that changes in the first level of such nested condition/decision structures do not change the coverage of the conditions and decisions on deeper level. In other words, it is assumed that the coverage of conditions or decisions on deeper levels will not be affected by the program transformation, and that the proof result is only dependent of the conditions and decisions of the first level.

- *Loop iterations.* At the moment only one iteration of a loop is mentioned as relevant path for analysis together with the possibility not to enter the loop. If the loop body includes structures with alternative control-flow, an appropriate number of paths representing one iteration of the loop with different control-flow inside the loop body is considered only. Different combinations of the alternative control-flow inside the loop in subsequent iterations are not considered at the moment.

- *Scoped path coverage.* Due to the complexity of possible segmentations, the *Mathematica* implementation of scoped path coverage is restricted. As described above, it is assumed, that both program fragments are completely enclosed by one segment.

4.8 Summary of Chapter 4

The implementation of the basic control-flow-graph structures described in Chapter 3 with the mathematical software system *Mathematica* is explained. In addition, the implementation of several proof functions which are able to automatically perform a coverage-preservation analysis is described.

Restrictions of the current implementation are listed.

CHAPTER 5

Use Cases

5.1 General Remarks

Introduction

One of the most important goals of the automatic generation of object code is to make the best use of the underlying computer technology and to produce code that is well tuned for a given architecture [39, 5]. The compiler should tailor the produced code to the specific architecture used, without requiring any modification of the source code. In general compilers attempt to be as aggressive as possible when doing optimizations. But they do it never at the expense of producing incorrect code. If there is no guarantee that an optimized operation produces no error, the optimization is omitted. It should be noted, that the term "optimization" is a misnomer, because most of the optimizations applied to a program do not result in object code which is optimal by any measure [40]. Optimizations may improve the performance of a program, but it is also possible that they decrease performance or leave it unchanged for the used set of input data.

Not all optimizations are performed at the same phase of the compilation process, and on the same level of intermediate language. Certain kinds of code optimizations are best done on a higher level of intermediate language, while others are best performed on a lower level. They are normally performed in a certain order to be as optimal as possible. In addition, not all code optimizations are done in a one-phase process. Some optimizations need some preparation steps first, intended to enable further transformation steps.

Code coverage analysis normally does not care too much about order or reasons why a certain optimization is done. The need to judge the influence of a code transformation is triggered by the fact, that the compiler does it. The only reason to take care about the conditions for a certain kind of transformation is to determine the properties of the transformation.

Structure of Use-Case Descriptions

Each of the use-case descriptions provided in this chapter is divided into four main subsections. The first subsection 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. The implementation subsection explains the transformation conditions and the basic facts for preservation analysis including aCFG models for the original and the transformed program. In the third part, the analysis section, some facts about the properties of the investigated code transformation are described, as far as they are important for the coverage-preservation analysis. Finally, the last subsection gives a summary of the results and provides some informal arguments, why a particular kind of coverage is preserved or not.

Note, that the explanations of the use cases presuppose two basic assumptions already considered in Chapter 3:

- The transformation of the examined program fragment does not change way, how the remainder of the program is executed. In particular, the transformations done in the investigated piece of code will not corrupt the code coverage achieved in the remainder of the program.

- The same execution paths will enter the transformed program fragment and the original program fragment. Especially, the input-valuations triggering the execution of the examined program fragment are the same for both versions of the fragment (Axiom 3.3).

Node identifiers and naming of input-valuation sets in the drawings of the aCFGs are chosen in conformance with the use cases in the code coverage *Mathematica* notebook. Additional circles are used to mark 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 thick dotted oval enclosing all conditions the decision is composed of. Functional relationships of nodes are drawn as dotted lines pointing from a node or hyper-node of the transformed program to the functional related node or hyper-node of the original program. Sometimes the string "$\mapsto X$" is placed beside a node of the transformed program, to point out that this node is identical with node X in the original program. Note, that for better understanding functional relationships are only drawn if they are important for analysis or important for better understanding of the use case.

Although compilers perform most of the optimizing code transformations normally on some kind of intermediate code, code examples are written in a pseudo source-code language for better readability. The examples presented as part of the description of a code optimization usually show a special application of the considered code transformation and do not claim to be a general description of the code transformation. However, the aCFGs are intended to draw the general case of the code optimizations. So there may be differences between the code example and the aCFG representation, especially concerning the interpretation of the transformation properties.

5.2 Useless Code Elimination

Description

Useless code, also called *dead code*, is reachable code which performs no computations that can affect final results [39]. A variable is called *dead*, if it is not used on any path from the location in the code where it is defined either to its redefinition or to the exit point of its scope. A statement is *dead*, if it computes only values that are not used on any path leading from the instruction. 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 [40].

Figure 5.1 presents on the left-hand side a code example for useless code which is removed by useless code elimination. 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 resulting in a transformed program shown on the right-hand side of Figure 5.1.

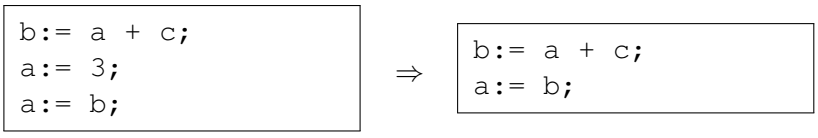


Figure 5.1: Code example for removing useless code. The first assignment to a has no effect, since its value is never used.

Implementation

Figure 5.2 shows aCFG representations for the original and the transformed program. Be aware, that each node of the graph can represent a single statements as well as a sequence of simple statements. Clearly, if the removed node represents a statement sequence, the whole sequence is assumed to be useless code. The transformation removes the useless statement node, and connects its predecessor with its successor directly. All other statements and their relative order with respect to control flow are preserved.

Analysis

Both versions of the program fragment consist of only one path without forks and joins. The only one path therefore includes all statements of the fragment and all input valuations triggering the execution of the examined program fragment will always execute each statement of the fragment.

More formal, there are only simple statements or statement sequences involved. Therefore all input-input-valuation sets inside each aCFG are equal. Furthermore, the equality $\delta_i = \varrho_j$ holds for all possible combinations of the indices i and j with $1 \leq i \leq 4$ and $1 \leq j \leq 3$.

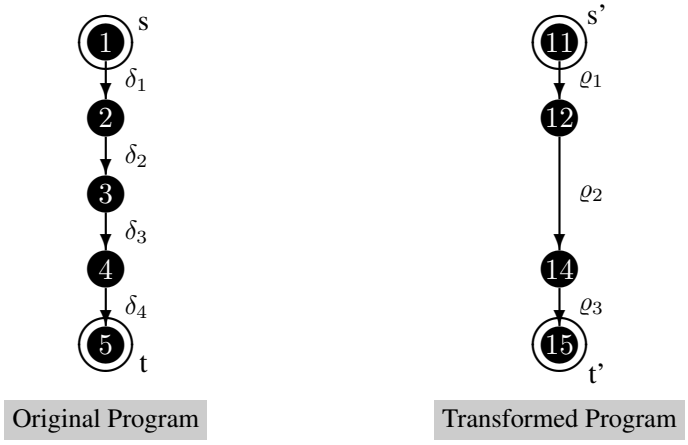


Figure 5.2: USELESS CODE ELIMINATION: aCFGs representing the program before and after applying useless code elimination.

Results

Statement Coverage is preserved, because all statements are included in the only one path and every execution of the fragment will also execute each statement.

The formal statement-coverage preservation condition is fulfiled, because all input-input-valuation sets of both aCFGs are equal.

Condition Coverage is preserved by default. Since both versions of the program fragment contain no condition, condition coverage is true by default.

Decision Coverage is preserved by default, because both versions of the program fragment contain no decisions, and therefore decision coverage is always true by definition.

Modified Condition/Decision Coverage is preserved by default, because of the absence of conditions and decisions.

(Scoped) Path Coverage is preserved, since the only one path of the fragment is executed whenever the execution of the program fragment is triggered.

5.3 Condition Reordering

Description

Condition reordering is an example of a code transformation that is not a classical optimization. It 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 common sub expressions. This enables the application of optimizing transformations like algebraic simplifications or common

sub expression elimination. Figure 5.3 presents a possible example, using commutativity of a logical *and* operator to reorder the conditions of a branch decision. It should be noted, that each of the operands can be a single condition or a logical expression treated as a unit.

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

⇒

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

Figure 5.3: Possible code example for a reordering transformation of two conditions inside a branch decision.

It should be noted, that condition reordering seems to be easy from the point of view of the source-code level but from the point of view of the program structure on object-code level or intermediate-code level it is often is not. The corresponding aCFG structure can become very complex depending on the type and number of Boolean operations and depending on the used evaluation semantics. In addition, priority rules and bracketing have also influence on the structure, because they can change the order in which conditions are processed. So it is not easy to find representative structures for the condition-reordering transformations.

On the other hand, semantic preserving reordering of conditions cannot be done arbitrary, because the priority of operations must be taken into account. Consider, for example, the expression $A \vee B \wedge C$ with A, B and C being conditions. This expression could be evaluated in two different ways:

1. From left to right: $(A \vee B) \wedge C$.
2. With the usual priority of the AND operation: $A \vee (B \wedge C)$.

These interpretations of the expression are not equal, because they will obtain different results for some value combinations of the three conditions. The reordering of the conditions must respect the evaluation sequence to preserve the semantics of the expression. For example, it is not allowed to just swap A and C, because this would produce a different expression. But it would be allowed to swap the expression enclosed in brackets with the single condition, for example, changing $(A \vee B) \wedge C$ to $C \wedge (A \vee B)$. But this kind of reordering is an equivalent transformation to that shown in the use-cases.

Implementation

The condition-reordering problem is demonstrated here with four instances of case studies. The first pair assumes branch statements with non-empty *then* and *else* branches, but with different evaluation semantics. The second pair analyzes branch statements with an empty *else* branch, but two different logical connectives for the conditions.

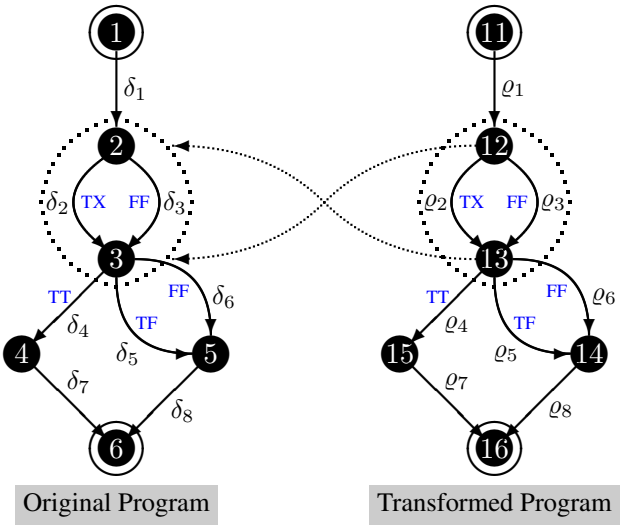


Figure 5.4: CONDITION REORDERING: Swapping conditions in a branch decision without short-cut semantics.

Full evaluated branch. The first use case in Figure 5.4 implements a full evaluated branch decision with two conditions connected by logical AND. The characteristic of a full-evaluated branch is, that different to the shortcut-evaluation semantic all conditions of a decision are always evaluated independent of the outcome of the already evaluated conditions. In the present use case, the second condition will always be evaluated independent of the outcome of the first condition.

Shortcut evaluated branch. The second use case shown in Figure 5.5 implements the same branch statement as described above, but with shortcut evaluation semantics (also called short-circuit evaluation). In this case, subsequent parts of a Boolean expression are only evaluated, if the evaluation done so far has not obtained a final result. To give an example, this behaviour corresponds to the semantics of the operators && and || in C/C++ and to the semantics of the operators *and then* and *or else* in ADA. In the present case the second condition is only executed if the outcome of the first condition reveals no final result.

Shortcut evaluated branch with empty *else*. These are two use cases demonstrating a short-cut evaluation implementation but with different logical connectives between the conditions of the branch decision. Changing the logical connective of the two conditions from AND to OR causes a relevant change of the structure of control flow. Compared with the non-empty *else* case, omitting the *else*-branch causes a lack of symmetry in the structure. This shows, that even minor changes in a program can have major influence on control flow with respect to preservation of code coverage.

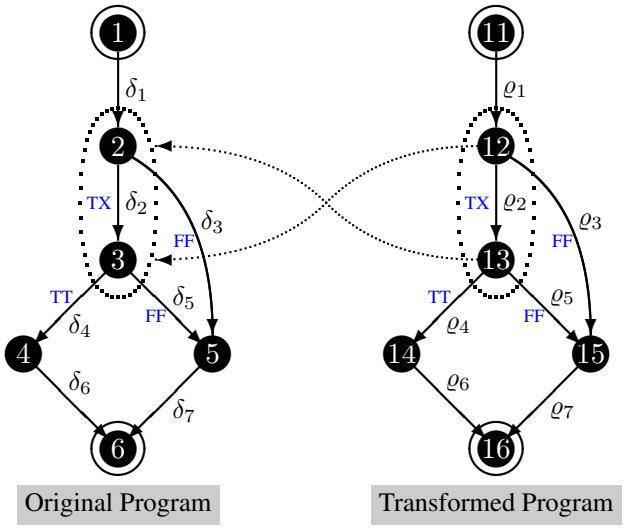


Figure 5.5: CONDITION REORDERING: Swapping the conditions of a branch with two conditions evaluated with short-cut semantics and a non-empty else fork.

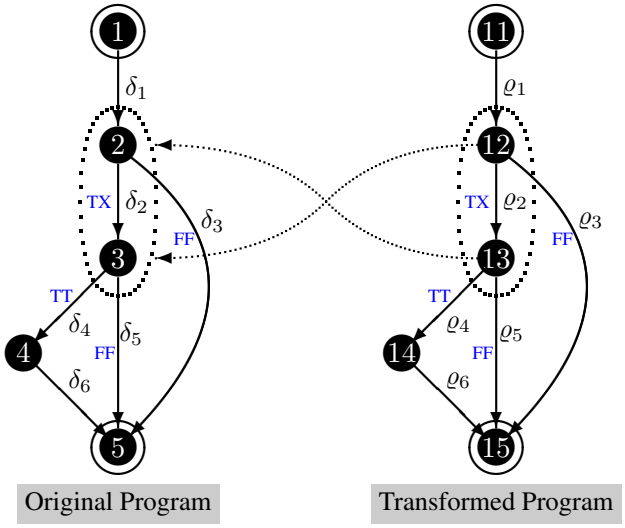


Figure 5.6: CONDITION REORDERING: Swapping conditions of a branch statement with two conditions using short-cut evaluation semantics. The *else* branch is empty and the conditions are connected with a logical AND.

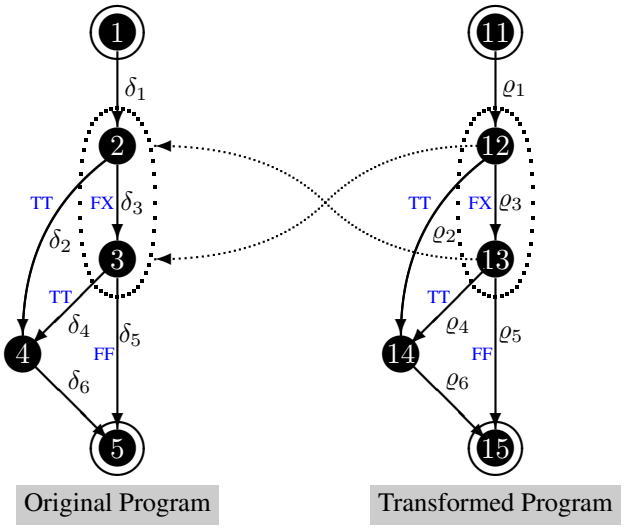


Figure 5.7: CONDITION REORDERING: Condition swap in a branch statement using short-cut evaluation semantics. The *else* branch is empty and the conditions are connected with a logical OR.

Analysis

The main characteristic of this 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. If the *else* branch is omitted, execution of statements inside the *then* branch must skipped for the same input values in both versions of the program fragment.

Full evaluated branch. Since the transformation just swaps the two conditions by leaving the evaluation semantic of the decision unchanged, the 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. Both equalities are detected by the automatic transformation relation determination, because the functional equivalences $12 \mapsto 3$ and $13 \mapsto 2$ imply the functional equivalence of the decisions.

All relevant transformation relations can be obtained with the automatic transformation-relation detection, using the simple method. There is no need to specify transformation relations manually.

Shortcut evaluated branch. In case of shortcut evaluation semantic, the connection $\langle 2, 5\rangle$ and $\langle 12, 15\rangle$ respectively can bypass the evaluation of the second condition. Therefore, 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 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

first value of the same test pairs is now assigned to condition 13 and the second value is assigned to the outcome of condition 12. Now the path $11 - 12 - 13 - 15 - 16$ is not covered anymore, because the test cases FF and TF bypass node 13 over the shortcut path. Condition coverage is therefore not achieved.

With similar arguments as in the full evaluated branch the relevant relations are determined to be $\delta_4 = \varrho_4$ for the *then* branch and $\delta_5 \cup \delta_6 = \varrho_5 \cup \varrho_6$ for the *else* branch.

Like in the full evaluated case, automatic transformation relation determination will be able to reveal these relations without manual intervention using the same node mappings as described above.

Shortcut evaluated branch with empty *else*. Characteristic relations derived from the transformations properties are similar to that for the non-empty *else* case:

- $\varrho_4 = \delta_4$, $\varrho_5 \cup \varrho_3 = \delta_5 \cup \delta_3$ for the version with AND connective, and
- $\varrho_2 \cup \varrho_4 = \delta_2 \cup \delta_4$, $\varrho_5 = \delta_5$ for the version with OR connective.

All the characteristic relations are detected by the automatic transformation relation determination using the simple method. If the *else* fork of a shortcut evaluated branch statement is empty, changing the logical connective of the conditions can change the preservation of structural code coverage. The result shows that statement coverage including condition is not preserved when changing the logical connective of the conditions from AND to OR. This happens, because of the different structure of the control flow. A branch structure with non-empty *else* fork is insensitive to changing the connective from AND to OR, due to its symmetry.

Results

Full evaluated 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 in the classical sense as well as in the enhance sense with taking care about conditions.

Condition Coverage is preserved, since all conditions are executed.

Decision Coverage is preserved, because this is the prerequisite for the validity of this transformation.

Modified Condition/Decision Coverage is preserved, because of the non-shortcut semantics no condition can be bypassed and so the unique-cause approach is applicable in the same way in both versions of the program.

(Scoped) Path Coverage is preserved. Because of the non-shortcut semantics all conditions inside the decision are on a unified single path (ref. to Section 4.3 for details about unified paths).

Shortcut evaluated branch with two conditions.

Statement Coverage is preserved for the classical view of statement coverage, without considering statement. If conditions are considered to be statements, then statement coverage cannot be preserved, because after swapping the conditions, the second condition may be bypassed and therefore not executed.

Condition Coverage is not preserved, because the shortcut semantics may redirect some paths and therefore reducing the set of input-valuations, the second condition decides on.

Decision Coverage is preserved, because this is a prerequisite for the validity of the transformation.

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

(Scoped) Path Coverage is not preserved, because short-cut semantics may redirect some paths.

Shortcut evaluated branch with empty *else* (AND).

Statement Coverage is preserved for the classical view of statement coverage, because the prerequisite for this transformation is to preserve decision coverage. Statement coverage is also preserved if conditions are treated like statements, because entering the *then* branch requires, that all conditions are executed and evaluate to *true*.

Condition Coverage is not preserved for the same reason as explained for the short-cut evaluated branch.

Decision Coverage is preserved, because this is a prerequisite for the validity of the transformation.

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

(Scoped) Path Coverage is not preserved, because short-cut semantics may redirect some paths.

Shortcut evaluated branch with empty *else* (OR).

Statement Coverage is preserved for the classical view of statement coverage, because the prerequisite for this transformation is to preserve decision coverage. Statement coverage is not preserved if conditions are treated like statements, because the structure of the decision allows bypassing the second condition when the *then*-branch is entered.

Condition Coverage is not preserved for the same reason as explained for the short-cut evaluated branch.

Decision Coverage is preserved, because this is a prerequisite for the validity of the transformation.

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

(Scoped) Path Coverage is not preserved, because short-cut semantics may redirect some paths.

5.4 Loop Peeling

Description

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 5.8. 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 [40]. 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 [5].

```
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
```

Figure 5.8: Example for application of loop peeling, taken from [5]. 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.

Implementation

The aCFG in Figure 5.9 implements 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 the copy of the loop body of the transformed program to avoid entering the loop, if the loop condition is not fulfiled at the beginning. The statement dealing with the iteration variable is assumed to be part of the loop body.

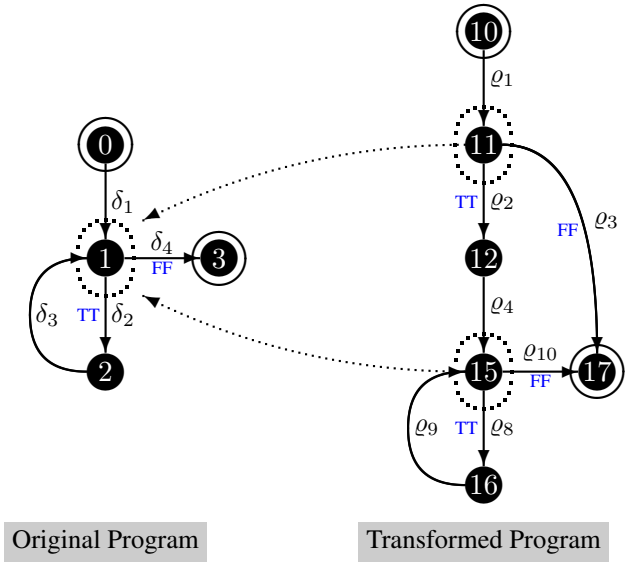


Figure 5.9: LOOP PEELING: aCFGs of a loop peeling transformation with $k = 1$ and a loop decision comprising a single condition.

Analysis

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 5.9). 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 exit equality (axiom 3.3 together with proposition 3.1). All these relations are found using the automatic determination function for transformation relations.

Results

Statement Coverage is not preserved, because it cannot be guaranteed that the loop in the transformed program is entered.

Condition Coverage is not preserved. Since it cannot be guaranteed, that the loop is entered it cannot be guaranteed that the second loop decision will be executed.

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

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

(Scoped) Path Coverage is not preserved, since the transformation creates new paths and so increases the number of paths to cover.

5.5 Loop Inversion

Description

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 [40]. 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 (Figure 5.10) and serves as basis for the analysis.

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

⇒

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

Figure 5.10: 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.

Implementation

The aCFGs for the loop-inversion transformation are presented in Figure 5.11. 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 $\langle 14 \rangle$.

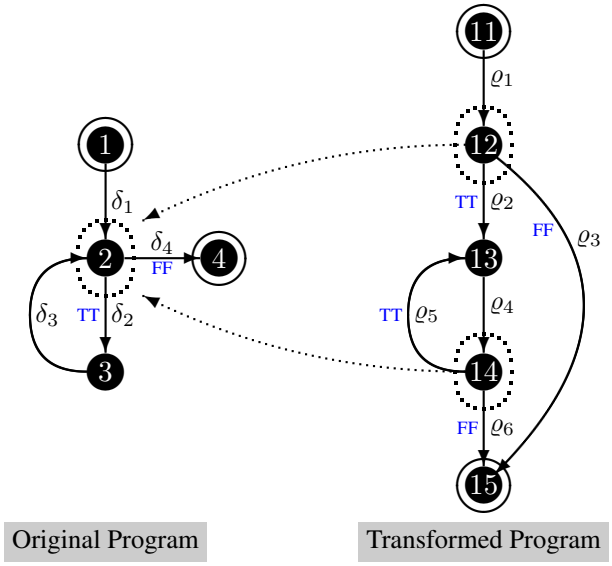


Figure 5.11: LOOP INVERSION: The aCFGs for a loop inversion transformation with an additional loops-closing test placed in front of the loop.

Analysis

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 mean 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 $\varrho_3 \cup \varrho_6$.

Most of the relevant relations are automatically detected by the simple relation determination method. The relation $\delta_2 = \varrho_4$ is not found automatically with the path-based method either. But this relation can be derived indirectly using the equality $\varrho_2 = \varrho_4$, which is found automatically with the path-based method.

Results

Statement Coverage is preserved, because the number of executions of the loop body must be kept unchanged by the transformation. So any test case that enters the loop of the original program will also execute the loop body in the transformed program.

Condition Coverage is not preserved, because the split of the loop-decision duplicates the conditions. There is no guarantee that the loop-back edge $\langle 14, 13 \rangle$ is ever used. But if edge $\langle 14, 13 \rangle$ is not used then condition 14 will never evaluate to *true*.

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.

(Scoped) Path Coverage is not preserved for a similar reason as described for condition coverage. The transformation increases the number of execution-paths and therefore the number of test cases achieving path coverage in the original program may be insufficient to achieve path coverage in the transformed program.

5.6 Loop Fusion

Description

Loop fusion, also called *loop jamming* [1], takes two adjacent loops that have the same iteration-space traversal and concatenates their bodies to create a single loop [40]. Loop fusion is possible, if the loops have the same iteration space and if there are no unfavourable dependences between the statements of the loop bodies. A possible example for this transformation is shown in Figure 5.12.

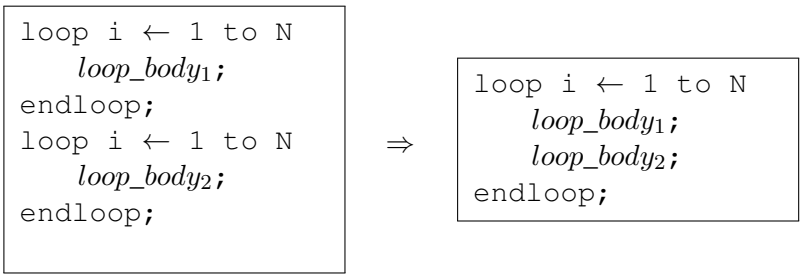


Figure 5.12: A pair of two loops with the same iteration space and the result of fusing them.

Implementation

As a prerequisite, loop fusion requires, that both loops in the original program execute exactly the same number of iterations for each possible input. On the other hand, the transformation must guarantee, that the number of iterations in the transformed program is the same than in the non-transformed version for each possible input.

Analysis

Clearly, this transformation preserves all kind of code coverage criteria, since the transformed loop decision has a semantics that is identical to the semantic of the original loops. The basic properties of the transformation allow derivation of the transformation relations $\delta_2 = \varrho_2$ and $\delta_5 = \varrho_3$. In addition, $\varrho_5 = \delta_7$ follows from the basic exit-equality requirement.

Not all these relations are found automatically, even by the path based method. But other relations found are sufficient to derive all missing relations indirectly.

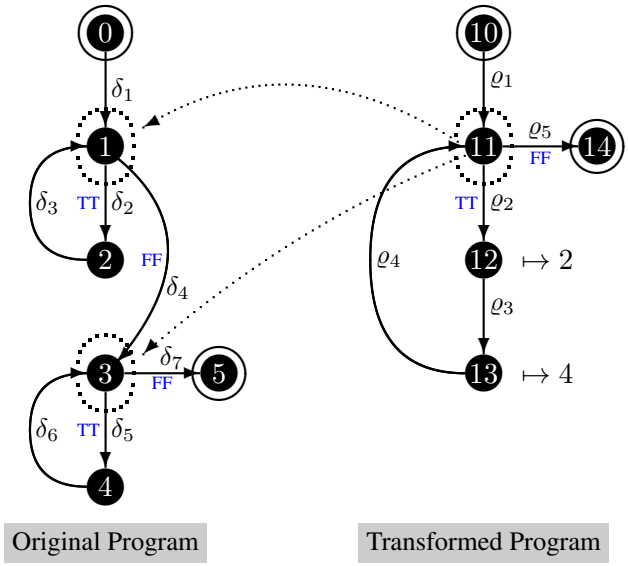


Figure 5.13: LOOP FUSION: aCFG representation of combining two loops with the same iteration space into one loop.

Results

Statement Coverage is preserved, because a valid loop-fusion transformation requires leaving the number of iterations triggered by each input-valuation unchanged.

Condition Coverage is preserved, since the loop-control of the transformed loop is identical to the loop-control of each non-transformed loop.

Decision Coverage is preserved for the same reason as condition coverage.

Modified Condition/Decision Coverage is preserved for the same reason as condition coverage.

(Scoped) Path Coverage is preserved since the number of paths decreases and the loop-control of the transformed loop is the same as the loop control of the non-transformed loop.

5.7 Loop Interchange

Description

Loop interchange swaps the order of the loop-decisions in a multiply nested loop [43]. The generalization of loop interchange allowing more than two loops to be moved at once is called *loop permutation* [38].

There is a broad range of reasons for a compiler to perform this transformation. Supporting parallelism, register reuse or spatial locality are some examples for the use of loop interchange [5]. Another reason may be to increase the amount of code executed between synchronization points. Finally, loop interchange is used as preparation step to vectorize array operations [39].

A simple code example for loop interchange is given in Figure 5.14.

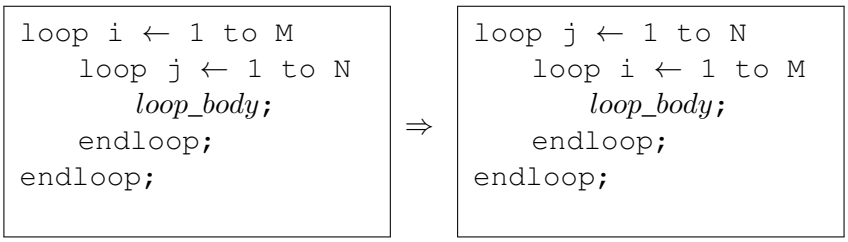


Figure 5.14: A double nested loop and the result of reordering the innermost and the outermost loop.

Implementation

The aCFG representation for the loop interchange transformation is presented in Figure 5.15. Statement and decision coverage can be proved using decisions composed of a single condition. But for such simple decisions condition coverage will trivially go with decision coverage, since there is a strict relation between the outcome of the decision and the outcome of the condition. A more complex decision structure with two conditions per decision is needed to be able to proof that condition coverage is not fulfiled by this transformation.

Analysis

The basic requirement of the loop interchange transformation is that the same input data must trigger the same number of iterations of the loop body in the original version as well as in the transformed version of the program. From the point of view of the code coverage analysis the input-valuations entering the innermost loop, δ_9 in the original version of the program and ϱ_9 in the transformed version, must be exactly the same. This is, because if only $\delta_9 \supset \varrho_9$ is true, then there must be some input valuation executing the loop body in the original program but not in the transformed version of the program. But this would violate the basic requirement of the transformation. The same argument can be considered in the other direction. Therefore the equality $\delta_9 = \varrho_9$ must hold for this transformation.

The automatic detection method for transformation relation is not able to determine the main relation $\delta_9 = \varrho_9$. So this relation must be supplemented manually. No transformation relation can be determined between δ_4 and ϱ_4, because the valuations passing the outer loop decisions can change in an unpredictable way.

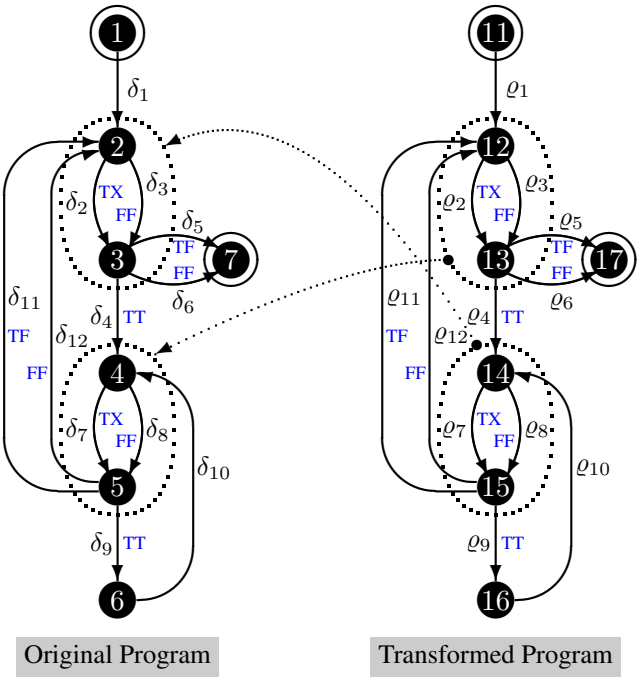


Figure 5.15: LOOP INTERCHANGE: Representation of the changes caused by swapping the loop decisions of a double nested loop.

Results

Statement Coverage is preserved. This is facilitated by the basic property of the transformation, because achieving statement coverage in the original program requires that the test-data trigger execution of the loop body. But then the loop body must be also executed by the same test-data in the transformed program.

Condition Coverage is not preserved. The inability of this transformation to preserve condition coverage is caused by the fact, that swapping the decisions can bring some more restrictive conditions in front. These more restrictive conditions possibly "catch" some input values, necessary to evaluate the conditions of the inner decision into all possible directions. Consider for example a condition in the innermost loop header evaluating to *false*, whenever the input value is an odd number. If all negative test cases comprise only odd numbers, swapping the decisions will bring this condition with its decision in front. Now all negative test cases will be rejected in the loop decision of the outer loop. As a consequence, the inner loop decision will only evaluate on the test-data provided by the positive test cases, and therefore some conditions of the inner loop decision in the transformed version of the program may become inactive.

Decision Coverage is preserved. This is a consequence of the basic loop structure. If the execution of the loop body was triggered by some test-data, the loop decision must be evaluated in either way. Otherwise there would be some non-terminating loop.

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

(Scoped) Path Coverage is not preserved for the same principle reason as condition coverage. Swapping the loop-decisions may possibly cause some paths of the original program to be joined after the transformation, leaving some other paths of the transformed program uncovered.

5.8 Loop Unrolling

Description

Loop unrolling replaces the body of a loop by k modified copies of the body and adjusts the loop-control code accordingly. The number k is called the *unrolling factor*. The original loop is called the *rolled loop* [40]. Unrolling reduces the overhead of executing the loop control decision. A possible code example for unrolling a loop with factor $k = 4$ is shown in Figure 5.16.

```
loop i ← 1 to N
    loop_body(i);
endloop;
```

⇒

```
loop i ← 1 by 4 to 4*⌊N/4⌋
    loop_body(i);
    loop_body(i + 1);
    loop_body(i + 2);
    loop_body(i + 3);
endloop;

loop i ← 4*⌊N/4⌋+1 to N
    loop_body(i);
endloop;
```

Figure 5.16: Example for unrolling a loop four times. The subsequent loop is created to catch remaining iterations of the rolled loop if the unrolling factor $k = 4$ does not divide the number of loop iterations evenly.

In the common case, the iteration counts of the loop are not always constants known at compile time and the unrolling factor does not divide the number of iterations evenly. Loops with a number of iterations unknown at compile time need therefore a rolled copy of the original loop following the exit of the unrolled loop. This additional copy of the loop executes remaining iterations, if their number is less than the unrolling factor. This approach is preferred rather than testing for early termination before executing each unrolled copy of the loop-body.

Implementation

Figure 5.17 shows the aCFG representation for the common case with an unrolling factor $k = 2$. The loop decision of the original program is composed of a single decision 2 which is sufficient, since it will reveal that decision coverage and condition coverage are not preserved. The unrolled copies of the body 3 of the original loop are represented by statements 13 and 14. The additional rolled copy of the loop body for executing possibly remaining loop iterations is represented by node 16. Hyper-node $\langle 12 \rangle$ represents the loop decision, modified for executing the unrolled version of the loop in double steps. Loop decision $\langle 15 \rangle$ is a modified copy of the original loop decision with shifted lower iteration limit.

Clearly instead of using a loop structure for processing the rolled copy of the loop body, a simple branch decision would be sufficient for $k = 2$. But the loop structure was chosen to visualize the common principle of this transformation.

The implementation of the aCFG uses the functional relationship symbols "-" for the *true* forks of the loop decisions. These symbols are associated to edges $\langle 12, 13 \rangle$ and $\langle 15, 16 \rangle$. They express the fact, that less execution paths than in the original program enter the loops of the transformed program. The loop decisions of the original and the transformed version of the program are functional-similar decisions. The unrolled loop deals only with multiples of the unrolling factor, omitting remaining iterations. On the other hand the rolled copy only handles remaining iterations omitting all iterations fitting into the multiples of the unrolling factor.

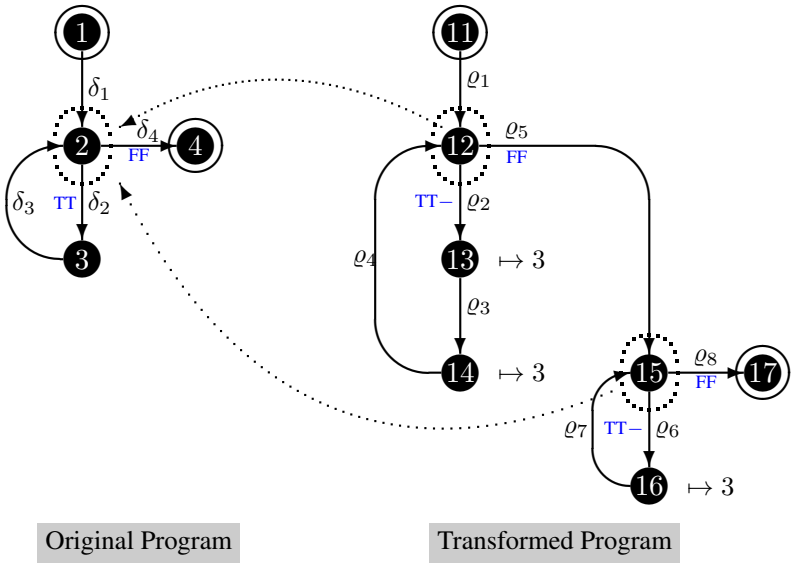


Figure 5.17: LOOP UNROLLING: Unrolling the body of the original loop 2 times and holding a rolled copy of the loop to execute remaining iterations.

Analysis

In the common case loop unrolling is not expected to preserve any kind of structural code coverage criteria. If the loop body is executed for some input valuation in the original program, the unrolled loop or the rolled loop will be executed a certain number of times. But there is no guarantee, that execution will enter both loops. In the best case the input-valuations entering one of the loops are equal, but in general they are a subset, which is stated by the functional-relationship symbol "-" on the *true* fork of the loop decision in the transformed program. The functional-relationship symbol directly implies $\varrho_2 \subseteq \delta_2$ and $\varrho_6 \subseteq \delta_2$. All necessary transformation relations can be derived completely from functional equivalencies, and no manual intervention is necessary.

Results

Statement Coverage is not preserved, since there is no guarantee, that each loop in the transformed program is ever entered.

Condition Coverage is not preserved, for the same reason as statement coverage. Since one of the transformed loops is possibly not entered, the conditions of the loop decision will not been evaluated with all possible outcomes.

Decision Coverage is not preserved, because otherwise this would contradict the fact, that statement coverage is not preserved.

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

(Scoped) Path Coverage is not preserved since there is no guarantee for executing all loops of the transformed program with the same test-data set as the original program.

5.9 Strip Mining

Description

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 [39, 43]. 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 [5]. A possible example for strip-mining the iterations of a for-loop is given in Figure 5.18.

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

⇒

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

Figure 5.18: Code example for strip mining: Dividing the execution of the left-hand side original loop into strips with a length of 2.

Implementation

The aCFG representations for automatic analysis are presented in Figure 5.19. The analysis uses complex decisions composed of two conditions. This is necessary to show that decision coverage is preserved while condition coverage is not. So, to get a more general result, the loop decision is composed of two conditions $\langle 2, 3\rangle$. The decision of the original program is distributed amongst two decisions $\langle 12, 13\rangle$ for the outer loop and $\langle 14, 15\rangle$ for the inner loop.

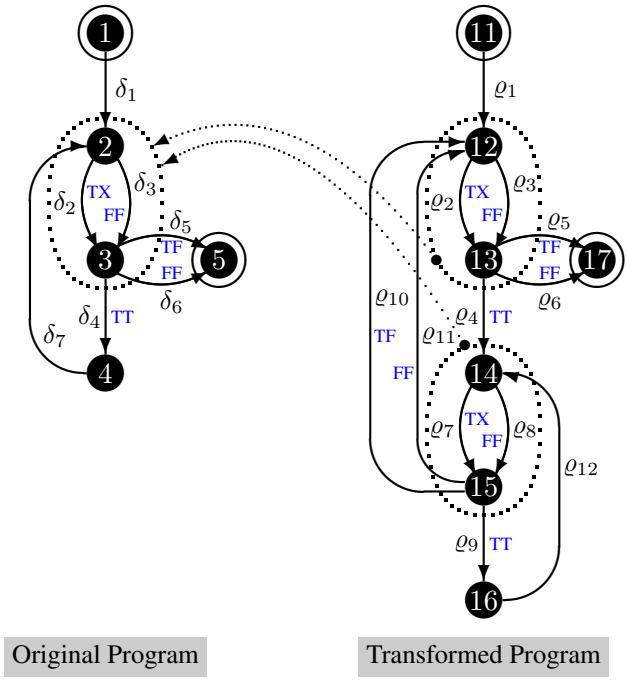


Figure 5.19: STRIP MINING: Dividing the iteration space of a loop to operate on smaller strips of the original loop.

Analysis

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.

Most of the essential transformation relations can be obtained automatically using the path-based method. On the other hand, the relation $\delta_4 = \varrho_9$ is not determined completely. If this relation is added manually, even the simple method is sufficient to get all relevant relations for analysis.

Results

Statement Coverage is preserved because the transformation is only allowed if the number of loop iterations is kept unchanged for the same set of input data.

Condition Coverage is not preserved. Due to the distribution of the decisions it cannot be guaranteed that the innermost decision is getting all input-values necessary to evaluate into all possible directions.

Decision Coverage is preserved. Since decision coverage in the original program requires entering the loop at least once, the same must happen in the transformed program and both decisions will therefore evaluate into all possible directions.

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

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

5.10 Loop Tiling

Description

Tiling is a transformation to improve the data locality of algorithms. Tiling reorders the execution sequence such that reused data is still in the cache or register file. This can cause a significant improvement on single processors as well as on multiple processors [56].

Tiling a single loop replaces it by a pair of loops with the inner one (called the *tile loop*) having an incremental like the original loop, and the outer one having an incremental appropriate for the lower and upper bounds of the inner loop. Tiling a loop nest of depth n may increases the depth of the loop nest anywhere from $n + 1$ up to $2n$, depending on how many of the loops are tiled. Tiling also interchanges the loops beginning from the tiled one inward to make the tile loops the innermost one in the loop nest. The number of iterations of the tile loop is called the tile size [40].

Tiling has several different names in the literature. Sometimes it is called "blocking" [5]. Other names are "strip mine and interchange" or "unroll and jam" [40]. Strip mine and interchange refers to the basic principle of tiling, because the loops are first divided to operate on strips of the original iteration space. Then the loops are rearranged such that they operate on a series of small polyhedrons of the original iteration space.

An example of tiling a double nested for-loop structure is presented in Figure 5.20 using a tile size of 2. The loops of the double nest $i \rightarrow j$ on the left-hand side of Figure 5.20 are first divided into a sequence $i \rightarrow i1 \rightarrow j \rightarrow j1$ of four nested loops. The step-size of the original loops i and j is adapted accordingly to the tile size of 2. In a second step, the $i1$-loop is interchanged with the j-loop to bring the tile loops together. The final result is shown on the right-hand side of Figure 5.20. The innermost loop nest is now operating on square tiles of size 2×2.

```
for i ← 1 to N do
  for j ← 1 to M do
    loop_body(i, j);
  endfor;
endfor;
```

⇒

```
for i ← 1 by 2 to N do
  for j ← 1 by 2 to M do
    for i1 ← i to min(i+1,N) do
      for j1 ← j to min(j+1,M) do
        loop_body(i1, j1);
      endfor;
    endfor;
  endfor;
endfor;
```

Figure 5.20: The starting point and the final result of tiling a double nested loop with a tile size of 2. Both loops are first divided by strip mining, then the loops $i1$ and j are interchanged to bring the tile loops inward.

Note, that tiling a single loop is the same as strip mining, described in Section 5.9.

Implementation

The aCFG representations used to analyze loop tiling is shown in Figure 5.21. The transformed programs aCFG is quite complex, because decisions with double conditions are needed to show, that condition coverage and decision coverage preservation are different. The final result is not surprising being aware that tiling combines two loop transformations: strip mining and loop interchange. Both of these transformations preserve statement and decision coverage but not condition coverage.

Analysis

The main characteristic of the transformation is the same as for other similar transformations, especially strip mining. The number of iterations of the inner loop must be kept invariant during transformation, and therefore the equality $\delta_9 = \varrho_{19}$ is mandatory. This relation must be specified

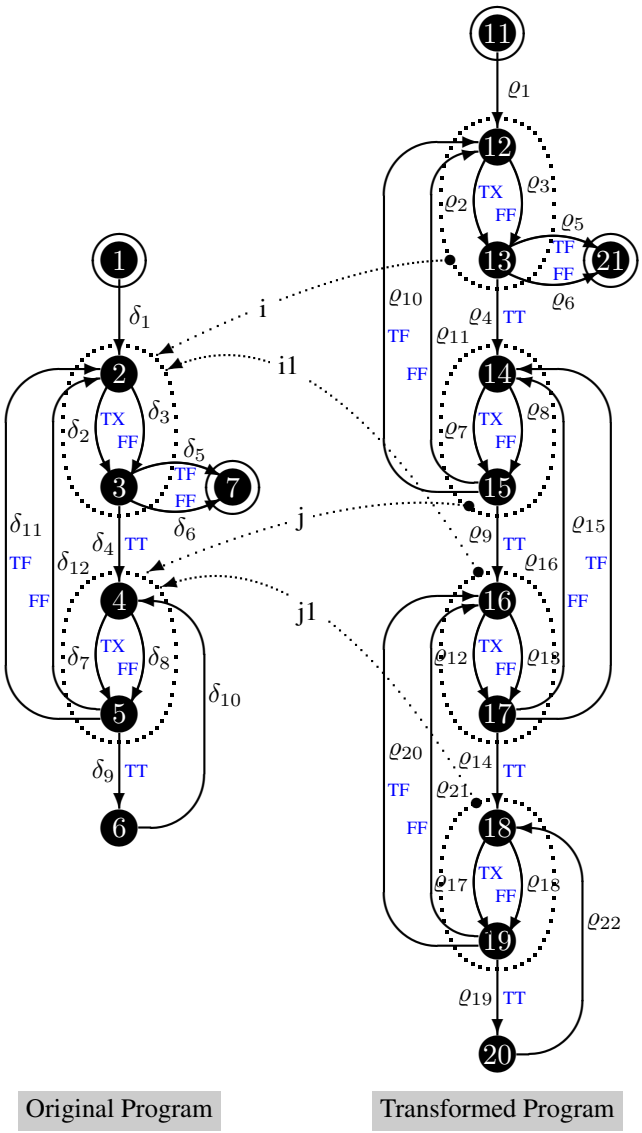


Figure 5.21: LOOP TILING: aCFG representation of tiling a double nested loop.

manually. All other relation can be determined automatically using the path-based method for determining valuation relations.

Results

Statement Coverage is preserved because the transformation must guarantee that the number of loop iterations is kept unchanged for the same set of input data.

Condition Coverage is not preserved. Due to the distribution of the decisions it cannot be guaranteed that the innermost decision is getting all input-values necessary to evaluate into all possible directions. Another aspect to mention is, that the increasing number of conditions may need more test-data to guarantee, that all conditions are evaluated into all possible directions.

Decision Coverage is preserved, because if decision coverage is achieved in the original program, then the loop must have been entered at least once. So the same must happen in the transformed program and all decisions will therefore evaluate into all possible directions.

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

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

5.11 Loop Unswitching

Description

Unswitching is a control flow optimization that pulls a loop invariant conditional branch out of the loop [40]. The example in Figure 5.22 assumes that the loops iteration variable is i, while the conditional branch depends on some variable k.

```
loop i ← 1 to N do
  if k ≠ 0 then
    then_branch;
  else
    else_branch;
  endif;
endloop;
```

⇒

```
if k ≠ 0 then
  loop i ← 1 to N do
    then_branch;
  endloop;
else
  loop i ← 1 to N do
    else_branch;
  endloop;
endif;
```

Figure 5.22: Example for pulling out a loop invariant branch predicate from a loop [40].

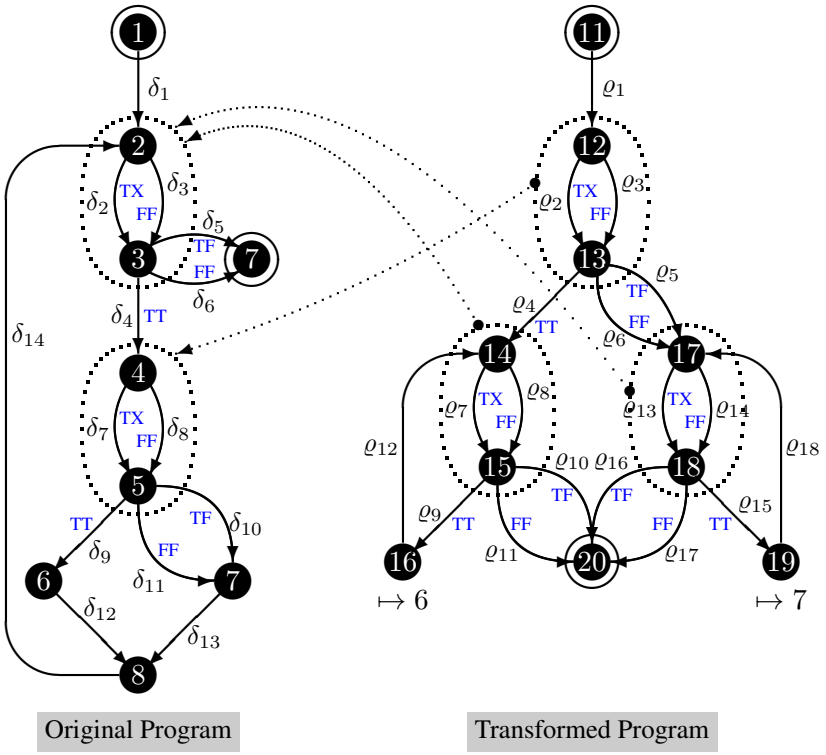


Figure 5.23: LOOP UNSWITCHING: Representation of the general case for pulling a loop invariant branch decision out of the loop.

Implementation

The aCFGs for automatic analysis are presented in Figure 5.23. The original loop decision is represented by nodes $\langle 2, 3\rangle$, while $\langle 4, 5\rangle$ represents the original conditional branch decision. Nodes 6 and 7 implement the statements in the *then* and *else* fork of the branch. Node 8 has neither a real function nor is it necessary for analysis, but it is useful here to produce a nicer drawing. If preferred, the loop back to node 2 could be drawn directly from nodes 6 and 7.

In the transformed version of the program, the conditional branch decision $\langle 12, 13\rangle$ has been pulled out of the loop, without changing the semantic. In addition, two copies of the loop decision, $\langle 14, 15\rangle$ and $\langle 17, 18\rangle$ have been placed inside the branch forks. Nodes 16 and 19 implement the corresponding statements taken from the *then* and *else* fork of the original branch.

Analysis

Basic relations for analysis are $\delta_{12} = \varrho_9$ and $\delta_{13} = \varrho_{15}$. Source of these relations is the requirement, that the *then* and the *else* statements must be executed in the transformed program the same number of times for each possible input than in the original program. Thus, statement coverage must be fulfiled.

On the other hand, the set of input data for the conditional branch in the transformed program is now bigger. It may contain additional values originally blocked by the loop decision. Therefore $\varrho_4 \supseteq \delta_9$ and $\varrho_5 \cup \varrho_6 \supseteq \delta_{10} \cup \delta_{11}$ must be true. Thus, decision coverage is preserved.

The transformation does not preserve condition coverage for the same reason as for similar loop transformations. The structural change only preserves execution paths of all input values triggering the execution of the statements located in the *then* and in the *else* fork of the conditional branch. Bit execution paths for all other input values may be redirected and therefore not reaching the same decision as in the original program. For that reason, conditions may be evaluated different and it cannot be guaranteed, that all conditions evaluate in either direction.

Results

Statement Coverage is preserved. Achieving statement coverage in the original program means that the loop has been entered at least once and the branch decision must have been evaluated into each possible direction. The number of iterations must be kept equal for the same set of input data during transformation. In addition, as a prerequisite the branch-decision must be independent of the loop-iteration variable. Thus, the branch decision in the transformed program will be taken in the same way as in the original program and each loop copy of the transformed program will be entered the same number of times. Therefore, statement coverage is also achieved in the transformed program with the same set of test data.

Condition Coverage is not preserved. Although the branch decision is presupposed independent of the loop iterations, it may filter out some input-valuations necessary for creating all possible outcomes of the loop decisions.

Decision Coverage is preserved, since the loop must be entered at least once and the branch decision must have gone into each possible direction. Because of the independence of the branch decision from the loop-iteration variable the branch statement in the transformed program must also go into each directions. If a outcome of the branch decision is taken, than the transformations must guarantee, that the number of iterations is the same for as in the original program for the same input-data set and so the loop decision must go into each direction.

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

(Scoped) Path Coverage is not preserved, because although the branch decision is presupposed to be independent of the loop-iteration variable it may redirect some paths necessary to cover all paths inside one branch.

5.12 Software Pipelining

Description

Software pipelining, also called *kernel scheduling*, is a preparation step to improve performance by utilizing parallelism at the instruction level. It reorganizes loops across loop iterations such that iterations are executed in overlapped fashion [3]. Software pipelining is an effective scheduling technique for VLIW processors. But its drawback is its complexity, since the problem to find an optimal schedule is NP-complete [35].

Software pipelining reorganizes a loop into three components:

1. A kernel including the code that must be executed on every cycle of the loop, once it has reached a steady state.

2. A prologue, which includes the code that must be executed before the steady state can be reached.

3. An epilogue to finish the loop, once the kernel can no longer be executed.

Software pipelining focuses on temporal movement of instructions through loop iterations, and not on spatial movement within a single loop iteration. Critical instructions whose results are needed early are moved to earlier loop iterations, so that their results become available within the current iteration. On the other side, instructions at the tail end of the critical path are moved to future iterations to shorten completion of the current iteration. In other words, the body of one loop iteration is pipelined across multiple iterations to take advantage of available resources within one iteration [5].

The code example in Figure 5.24 shows a loop with two statements, both dependent on the loop iteration variable i. The transformed program on the righthand side of the figure assumes, that N is always greater than 1. The first iteration of the first loop statement has been peeled out of the loop to serve as prologue, while on the other side the last iteration N of the second statement is pulled out and placed behind the loop to serve as epilogue. The kernel reorders execution of the statements of the original loop, such that the $i-1$'th iteration of the first statement is combined with the i'th execution of the second statement [29].

```
loop i ← 1 to N
    loop_statement_1 (i);
    loop_statement_2 (i);
endloop;
```

⇒

```
loop_statement_1 (1);
loop i ← 2 to N do
    loop_statement_2 (i − 1);
    loop_statement_1 (i);
endloop;
loop_statement_2 (N);
```

Figure 5.24: Code example for software pipelining with the original program on the lefthand side and the pipelined version of the program on the righthand side [29].

Implementation

The behaviour of software pipelining, concerning structural code coverage is quite similar to other loop transformations like *loop peeling* for instance. The structural changes performed during transformation must guarantee for each input value that the statements of the loop body in the transformed program are executed the same number of times as in the original loop.

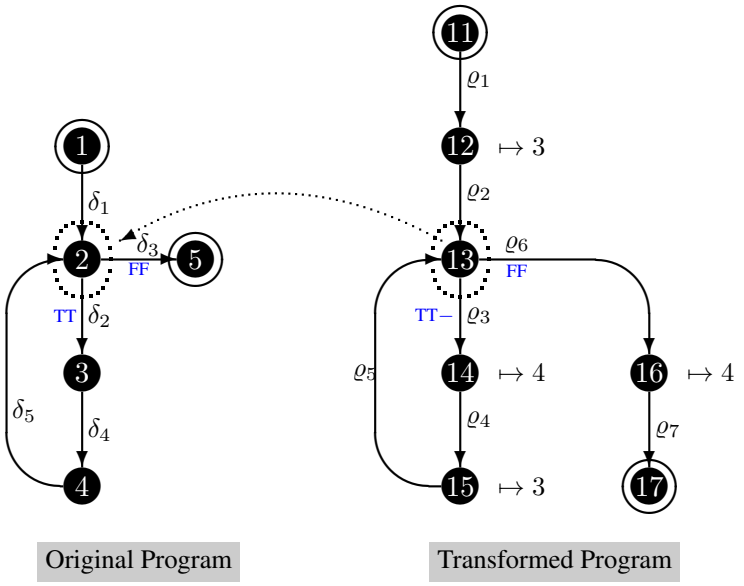


Figure 5.25: SOFTWARE PIPELINING: Distributing the execution of the statements 3 and 4 of two successive iterations of the loop to create an overlapped execution of iterations.

Figure 5.25 represents a common case for this loop transformation, with nodes 3 and 4 representing two statements that should be reorganized for overlapping execution. The used structure is inspired by the source code representation for software pipelining used by Kirner in [29]. In the pipelined version, node 12 represents the prologue (executing the statement represented by node 3 in the first iteration of the original loop), and node 16 represents the epilogue (executing the statement represented by node 4 in the last iteration of the original loop). The pair of nodes 14 and 15 implements the kernel. Decision $\langle 13 \rangle$ and decision $\langle 2 \rangle$ are classified as functional similar. Less execution paths than in the original loop will enter the transformed loop, because of the adapted loop bounds. This fact is taken into account by tagging the outgoing edge $\langle 13, 14 \rangle$ of the transformed decision with the functional-relationship label "−" to support the automatic transformation-relation determination procedure.

Analysis

The following example should give an informal insight about the effect of this transformation on structural code-coverage. Consider, that the test of the piece of code in Figure 5.24 has been performed with a set of test data triggering one iteration of the loop of the original program. This test will be able to achieve all kind of structural code-coverage in the original program. After

the transformation, executing the prologue and the epilogue of the program fragment will satisfy the transformation condition, that the first iteration must be executed for both statements of the original loop body. But the loop kernel will never be entered, because this happens only if two or more iterations of the loop are triggered by the test data.

From a more formal point of view, the main crunch is the relation $\varrho_3 \subseteq \delta_2$, which prevents the required superset relations for $IV_R\,(14)$ as well as of $IV_T\,(13)$.

Results

Statement Coverage is not preserved, because there is not guarantee that the loop of the transformed program is ever entered. Thus the statements inside the loop are not guaranteed to be executed.

Condition Coverage is not preserved, since the loop of the transformed program may not be entered.

Decision Coverage is not preserved, because the loop of the transformed program may not be entered and therefore the loop decision will never evaluate to an outcome necessary for entering the loop.

Modified Condition/Decision Coverage is not preserved since condition coverage and decision coverage are not preserved.

(Scoped) Path Coverage is not preserved. Since the loop is not guaranteed to be entered, the path through the loop may be never executed.

5.13 Branch Optimization

Description

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 [37, 40].

Implementation

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 5.26. 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, shown on the righthand side of Figure 5.26 is that the redirection possibly leaves the former branch instruction 13 unconnected.

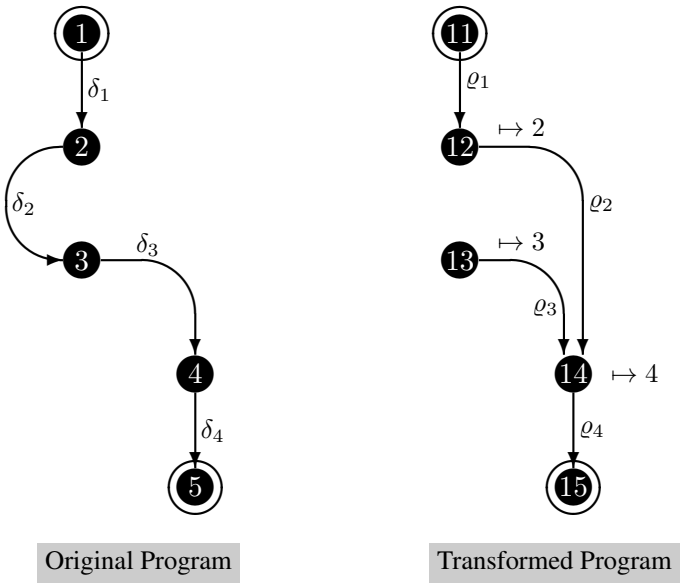


Figure 5.26: BRANCH OPTIMIZATION: Redirecting a branch to a branch instruction directly to the branch target.

Branch optimization is an example showing some problems concerning the approach to analyze single optimizations. The optimization obviously reduces the reachability of the bypassed branch 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 is completely bypassed, the following dead code elimination will re-establish statement coverage again.

Analysis

This use-case shows some limitations of the chosen model and its implementation. The classification *true* for preservation of coverage criteria other than statement coverage results from the default behaviour due to the absence of conditions and decisions. The implication $DC \Longrightarrow SC$ is obviously not correct concerning the program fragment, because it contains no conditions and decisions. But the default behaviour makes sense, because these coverage criteria are assumed to be unchanged in the rest of the program. On the other hand, 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* 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.

Results

Statement Coverage is not preserved, since the reachability of the first branch statement will be reduced or the statement will become unreachable.

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

Decision Coverage is preserved by default, because the program fragment contains not decision.

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

(Scoped) Path Coverage is classified as preserved by default. The reason for this behaviour is, that it is assumed that node 13 in Figure 5.26 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.

5.14 Final Remarks

Elaborating the use-cases has shown, that the chosen aCFG representations of program fragments are very intuitive to handle, because most of the aCFG structure can be directly derived from the program structure. On the other hand, it is not a general approach with respect to the structure of the investigated program fragment. Changes in the program, which seem to be small on source-code level can change the structure of the corresponding aCFG dramatically, creating the need to open a new use-case to proof the examined behaviour. Example for this fact are discussed in Section 5.3, where adding brackets or changing the logical connective of conditions can create a new structure of a decision. The evidence of the result of a use-case therefore depends on the ability of the use-case developer to find a representative structure for a program transformation, which is as general as possible.

An aspect that has also been shown with the condition-reordering problem in Section 5.3, is that a less complex structure can reveal surprising results with respect to coverage preservation. In some cases the less complex structure is more restrictive with respect to preserving a certain kind of code coverage, in other cases it may be less restrictive.

Another inherent problem related to the approach of analyzing single transformations is the fact, that subsequent code transformations may reverse the effect of a single transformation in the sequence. In other words, two sequent applied code transformations may as a whole preserve the investigated code coverage, even if each of them does not preserve the investigated code coverage. For example, the investigations on the use case of Section 5.13 have been shown, that branch optimization with a subsequent dead-code elimination can re-establish statement-coverage.. So analyzing only single code transformations reveals sometimes an overcautious

result for a sequence of code-transformations. The only way to deal with such configurations is to create a model for the final result of the transformation sequence. But in practice it is hard to identify cases where this is necessary.

5.15 Summary of Chapter 5

Several common optimizations from the literature are explained in this chapter. For each optimization an informal description and aCFGs for the original program and the transformed program are presented. In addition, each use case is supplemented with the results of the coverage-preservation analysis.

General things learned from the use cases are noted at the end of the chapter.

CHAPTER 6

Summary

6.1 Conclusions

The work related to this thesis has established formalism for applying the structural-code-coverage preservation criteria described in [31] on structural changes of fragments of programs. The formalism described in Chapter 3 is based on a kind of control-flow graph representation, called the *analysis control flow graph* (aCFG). The aCFG model, introduced in Section 3.1, is used to create formal representations of fragments of the examined program, representing the part of the program where the program transformation is done.

Some existing approaches to formally specify program transformations and to describe the effect of a program transformation have been discussed in Section 3.3. Usual formal specifications of programs widely used for proving their correctness have been shown to be not very helpful for coverage preservation analysis. They are often closely related to the result values of operations, classifying two pieces of a program to be identical, if they deliver the same result values for the same input. But they offer no direct indication about the changes of the program structure caused by the transformation. The same applies to the transformation conditions expressed, for instance, in some terms of temporal logic and used to formally proof the correctness of code optimizations.

A mathematical system has been established in Section 3.2 and Section 3.4 to obtain relationships between input-valuations, necessary to apply the coverage-preservation conditions described in Chapter 2. Graph transformations, often used as formalism to describe structural changes of programs, have been mentioned to derive changes of execution-paths automatically. But execution path relations between the elements of the graph are often distorted or completely lost during the graph transformation process. This is the reason, why the method established in this thesis uses a "post-mortem" analysis, based on two models of the examined program fragment. One model is representing the examined fragment of the original program and the other represents the same fragment of the program after the transformation. Based on functional relationships between statements of the original program and statements of the transformed program the necessary relations for analysis are obtained.

The established formalism served as a basis to implement a framework in the mathematical software-system *Mathematica*. Some details of the implementation of this framework have been described in Chapter 4. The framework is able to do most of the steps for coverage-preservation analysis automatically. An automatic execution of coverage-preservation proofs has been implemented (see Section 4.5) for statement coverage, decision coverage, condition coverage and modified condition/decision coverage. Scoped-path coverage has only been implemented in a restricted version, which avoids the complexity of segmentations of the program that cut the examined program fragment.

The results achieved with the *Mathematica* framework, processing a certain number of use cases in Chapter 5, have shown that code optimizations in particular and code transformations in general can be analyzed automatically based on the structural changes represented by aCFGs. One lesson learned from this work is the fact, that apparent small changes in the structure of a program can have severe effects on the preservation ability of a code transformation with respect to a particular coverage criterion. An example for that, shown in Section 5.3, is changing the connective of two conditions from AND to OR in a shortcut evaluated conditional branch, which changes statement coverage preservation for the conditions.

One drawback of the taken approach, discussed in Section 5.14 is, that it is not a general approach with respect to the structure of the investigated program fragment. Sometimes it is hard to find the representative structure for a program transformation to get a general statement about the ability of the transformation to preserve a particular code coverage. Another possible disadvantage is the fact, that program transformations are analyzed with a restricted view to single transformations. The estimation achieved with this restricted view may be too pessimistic in some cases, because subsequent program transformations can sometimes compensate the effect of prior transformations, as shown in Section 5.13.

6.2 Results

Several common code optimizations and transformations described in the literature have been examined with respect to their ability to preserve the structural code coverage criteria under discussion. The results of these investigations are presented in table 6.1. One column is spent for each kind of code coverage. The entry ✓ in a column means, that the certain kind of coverage is preserved. The column entry · indicates that the automatic analysis failed to proof the particular preservation condition.

The first two columns contain the results for statement coverage, where *SC w/o C.* contains the result, when conditions are not treated like statements (statement coverage is only checked for nodes in B). *SC w. C.* presents the results when conditions are treated like statements (statement coverage is checked for $B \cup C$). The columns *CC*, *DC* and *MCDC* store results for condition coverage, decision coverage and modified condition/decision coverage. Finally *PC* is the result for checking preservation of a restricted kind of (scoped) path coverage, as described in section 4.5. The order in which the transformations are listed is related to the description in chapter 5.

The results of the detailed manual and automatic analysis worked out in this thesis have been compared with the results given in [30]. The table presented there was a first estimation of the

preservation properties of certain code optimizations. Most of the results have been confirmed, but for some code optimizations the classification has been revised based on the results of this work. The changed entries are marked in table 6.1 with a gray background.

	SC w/o C.	SC w. C.	CC	DC	MCDC	PC
Useless Code Elim.	✓	✓	✓	✓	✓	✓
Full Evaluated Branch	✓	✓	✓	✓	✓	✓
Shortcut Branch	✓	✓	·	✓	·	·
Empty Else (AND)	✓	✓	·	✓	·	·
Empty Else (OR)	✓	·	·	✓	·	·
Loop Peeling	·	·	·	·	·	·
Loop Inversion	✓	✓	·	·	·	·
Loop Fusion	✓	✓	✓	✓	✓	✓
Loop Interchange	✓	✓	·	✓	·	·
Loop Unrolling	·	·	·	·	·	·
Strip Mining	✓	✓	·	✓	·	·
Loop Tiling	✓	✓	·	✓	·	·
Loop Unswitching	✓	✓	·	✓	·	·
Software Pipelining	·	·	·	·	·	·
Branch Optimization	·	·	✓	✓	✓	✓

Table 6.1: Result of the automatic analysis of certain kinds of code transformations and code optimizations with respect to structural code coverage preservation.

Implementing the automatic analysis, two different methods have been developed to support the automatic setup of relations between the original and the transformed version of the analyzed program fragments (refer to Section 4.4). The first method is only based on a local view of the relations inside a control flow graph. It is very simple, easy to implement and efficient. The second method is based on execution paths, which adds a more global view to control flow analysis but also adds more complexity due to the exponential growth of the number of paths.

Practical experience with the use cases has shown, that the more complicated path based method sometimes is able to gain some more relevant relationships than the simple method. But even using the more complicated method, in most cases manual specification of relations relevant for the investigated transformation is mandatory. An example is the loop interchange use-case provided in Section 5.7.

6.3 Future Work

The use cases, presented in this thesis are based on descriptions of code transformations found in the literature. So one logical continuation of this work would be to analyze real compilers to learn in particular how they change program structure when optimizing code during the compilation process. Another extension of this work would be the experimental integration of the compilation profiles created by the automatic analysis into a real compiler.

Related to the analysis method itself, there are several extensions conceivable, in the established formalism as well as in the *Mathematica* implementation. Here are some examples for that:

- Investigating nested structures of conditions and decisions.
- Releasing some of the limitation of the model and/or implementation, like
  - implementing coupled conditions,
  - providing the possibility to implement more complicated decision structures,
  - extending the model to support decisions with more than two outcomes, allowing to create aCFG models of, for example, `switch`-statements in C/C++,
  - extending the aCFG to be able to create models for parallel executed program fragments, for example, to create models of optimizations distributing task on parallel processors,
  - creating more general graph elements, representing repeated sub-structure for instance.
- The consequences of the decision to take only one iteration of a loop with all possible variants were not studied in detail.
- Extending the automatic search for relationships inside the analysis control flow graphs to get better results and possibly avoid the need to specify manual relations. Especially, creating a possibility to gain the transformation relations from some kind of structure related formal transformation description.

Finally, supplementing the work with analysis of further optimizations would be another beneficial direction for future work. An interesting aspect may be to examine sequences of transformations, where subsequent steps re-establish the preservation conditions compromised by earlier steps.

APPENDIX A

Mathematica, a Short Survey

This chapter should provide a brief description of the used *Mathematica*, programming constructs. It is intended to help readers not familiar with *Mathematica* to read and understand the details of the *Mathematica* implementation.

A.1 Introduction

Mathematica is a fully integrated environment for technical computing [58]. It is a modular system where the front-end interacting with the user, is separated from the *kernel*. The *Mathematica* kernel actually performs computations and may be run either on the same computer as the front-end or on a different computers. The communication between kernel and front end is handled by *MathLink*, using any available networking mechanism.

The most common type of front end for *Mathematica* is based on an interactive document called *notebook*, and is supported on most computer systems. Notebooks are interactive documents mixing *Mathematica* input and output with text, graphics and other material. Although there are slight differences on different computer systems, the structure of *Mathematica* notebooks is the same on all computer systems. If a notebook is created on one computer system it should be immediately useable on another computer system. Especially the commands given to the *Mathematica* kernel are absolutely identical on every computer system.

A *Mathematica* notebook is a structured interactive document organized into *cells*. A bracket on the right of the screen display indicates the extent of a cell. Each cell contains material of a definite type like text, graphics or a *Mathematica* expression. The prepared contents of a cell can be sent to the kernel by pressing SHIFT-RETURN or SHIFT-ENTER. The kernel processes the material and sends back whatever output is generated. The front end will create new cells in the notebook to display this output. However, if the material sent to the kernel is finished with a `;` character, the output is suppressed. This is sometimes useful, if the operation reveals a trivial result.

Each cell within a notebook is assigned a particular style to indicate its role within the notebook. A style specifies a whole collection of properties for a cell. It not only defines the

format of the cell contents but also their placement and spacing. Material intended as input to be executed by the kernel for instance is typically in a style called *input style*. Larger notebooks commonly have chapters and sections, each represented by a group of cells. A bracket on the right indicates the extent of these groups. Groups of cells can be open or closed. If a group is open, all it's cells can be seen. If it's closed, only the heading cell in the group is shown.

If a notebook document is opened, nothing of its contents is normally sent to the kernel for evaluation, until the user explicitly requests it. Cells within the notebook can be identified as *initialization cells*to be processed automatically in top to bottom order. Note, that the code coverage notebook emphasizes initialization cells containing function definitions with a yellow background colour. The situation when the initialization cells will be processed depends on the current preference settings of the *Mathematica* front-end. These are some of the possibilities to handle initialization cells:

- They are executed whenever the notebook document is opened.
- The user triggers the execution of the initialization cells explicitly by choosing the menu item "Kernel→Evaluation→Evaluate Initialization".
- If a notebook document contains any initialization cells and if these cells where not executed by one of the procedures above, the front-end presents a pop-up the first time a command is sent to the kernel asking the user whether or not the initialization cells should be executed.

A certain amount of mathematical and other functionality is built into *Mathematica*. But *Mathematica* is an extensible system and it's always possible to add more functionality. These additional functions are included into *Mathematica packages* containing collections of definitions for particular application areas. The code coverage analysis system only loads and uses the package `« DiscreteMath`GraphPlot`;` [49] for drawing simple pictures of the specified aCFGs. Be aware, that the work of this thesis uses *Mathematica* version 5.2, and that this package has changed in newer versions [57].

A.2 Expressions, Variables and Functions

Many different kinds of objects like mathematical formulas, lists and graphics are handled in *Mathematica*. Although they may look very different, they are represented in one uniform way as *expressions*. Expressions are often used to specify operations or functions but they can also be used to maintain user-defined structures. Expressions are built from atomic blocks (numbers, strings or symbols) or from other expressions. *Numbers* are sequences of digit's representing a numerical value of a certain type like integer or real. *Strings* are sequences of characters representing arbitrary text.

An expression may be a single expression or a compound expression. A *compound expression* is a sequence of single expressions separated by semicolons. It can be placed wherever an expression is required. Compound expressions are evaluated in top to bottom (left to right)

order, corresponding to the control flow[1]. Its result is the result value of the last expression in the sequence. Putting a semicolon at the end of an expression or expression sequence is like giving an empty statement to the end of a sequence. The empty expression always returns *Null*. So ending a *Mathematica* input with a semicolon suppresses the output of the operation.

A *symbol* is a name for a *Mathematica* object, and it can serve many different purposes. It can be a variable just standing for itself, or it can be the name of another expression. Much of the flexibility of *Mathematica* comes from its ability to mix these different purposes. Different to traditional programming languages, variables in *Mathematica* can be used in a formal symbolic calculation as well as for numerical calculations. Variables like `x` can be treated in a formal symbolic fashion, but if needed `x` can be replaced by a definite value. Once a variable is defined, the definition will stay, until it is redefined or explicitly removed.

A symbol is created with its first use and needs no explicit declaration. It has global validity. As long as a symbol is not defined, it is treated in a formal symbolic way. A symbol is defined by assigning a value or an expression to it. When assignments are made to symbols, the two different types of assignments in *Mathematica* have to be carefully distinguished:

Immediate assignment is written `Set[`*lhs,rhs*`]` or more commonly *lhs*=*rhs*, where *lhs* stands for the left-hand side and *rhs* stands for the right-hand side. *Mathematica* evaluates the right-hand side immediately when the assignment is made.

Delayed assignment is written `SetDelayed[`*lhs,rhs*`]` or more commonly *lhs*:=*rhs*. In this case the right-hand side will not be evaluated until the left-hand side is used inside an expression.

Usually the delayed assignment is used for defining functions, because these are intended to be evaluates when used inside an expression and not when they are defined. In contrast, immediate assignment is normally used to assign a concrete value to a variable. The value, assigned to a symbol is permanent until the end of the current *Mathematica* session. A symbols value can be redefined by using the symbol again on the left side of an assignment. The value defined for a symbol can be removed anytime with `Clear[`*symbol*`]`.

In addition to built in functions, *Mathematica* allows the definition of user defined functions that work like mathematical functions, operating on specific expressions and output unique expressions for each input [53]. The left-hand side of a function definition is a symbol followed by an argument list enclosed in square brackets. The right-hand side can contain an arbitrary sequence of expressions, possibly including other functions, which should be evaluated when calling the function.

Mathematica provides several ways to create user-defined functions. The coverage analysis notebook uses two of them:

- Simple functions are defined by writing the expression, the function is composed of, to the right-hand side of the function definition. Sequences of expressions can be used as well, separating the single expressions by semicolons.

[1] There are several commands for changing the control flow like `Return` or `Break`. The code coverage notebook makes no use of these operations. More information about control flow changing commands can be found in the *Mathematica* online manual.

```
gB[G_] := G[[1]]
```

is an example for a simple function definition with one argument.

- Setting up a function as *module* allows to create a more complex procedure with a list of variables to be treated as local variables.

```
gReplaceNode[e_, iv_, iw_] := Module[{head = gEdgeHead[e],
   tail = gEdgeTail[e]},
    If[head == iv, head = iw];
    If[tail == iv, tail = iw];
    gReconnectEdge[e, head, tail]
 ]
```

is an example for a function definition with 3 arguments `e`, `iv` and `iw` as module with two local variables `head` and `tail`. The list of local variables is enclosed in curly brackets. Note, that local variables can be initialized together with their declaration. This method is equivalent to just declaring a local variable and assigning a value later.

The behaviour of a local variable is similar to other variables, with the only difference that the symbol is only valid inside the module. Especially, if a local variable is not defined when it's used inside an expression it is treated in a formal symbolic way.

Functions, independent whether they are user-defined or built-in, are used similar to standard mathematical functions. They are called by writing the function name followed by the list of arguments enclosed in square brackets[2]. Used inside an expression, the result of the function replaces the function call for further evaluation. When creating a user-defined function, the result of the whole function is simply the result of the last expression evaluated on the right-hand side of the function definition.

An underline character must follow each argument name on the left-hand side of a function definition. This advices *Mathematica* to replace each occurrence of the arguments name on the right-hand side with the actual expression of the argument. Defining `f[x_]` for instance means, that each occurrence of the symbol `x` on the right-hand side of the function definition is replaced by the actual value of the argument `x_` when calling `f`.

Note, that *Mathematica* allows defining multiple definitions of the same function. Multiple definitions of a function differ in their number or form of arguments given to that function. The code coverage analysis notebook uses this feature to define functions having optional arguments. A second way to define multiple versions of a function is it's piecewise definition. Piecewise defined functions have a range specification appended to each version of the function. The range specification defines the condition, for which range of their arguments a certain version of a function should be called.

[2] *Mathematica* allows to write a multiplication in standard mathematical notation, just writing the operands in a sequence. When using round brackets for function calls, the problem arises to distinguish the multiplication $f(a+b)$ "multiplying the variable f with the expression $(a+b)$" from the function-call f$(a+b)$ "calling function f with arguments $a+b$".

Commentary text inside *Mathematica* programming code can be added at any point of the code. It is enclosed in matching commentary-brackets `(*` and `*)`. Comments are not restricted to a single line and can be nested in any way [58].

A.3 Lists, Tuples and Sets

Constructing, Accessing and Measuring Lists

Mathematica provides no special set or tuple data-types for representing collections of objects. Instead it uses a fundamental data structure called *list*. A list groups objects together and uses them as a single entity. The elements of a list can be expressions of different types, possibly other lists [58, 53]. The empty list is represented by the expression `{}`.

Lists can be created in various ways:

`{`$elem_1, \ldots, elem_n$**`}`** is one of the most common forms for constructing a list of length n, comprising the expressions $elem_1, \ldots, elem_n$. A sequence of integers enclosed in curly braces, for example `{1,2,3,4,5}`, constructs a list of 5 integer numbers. As an alternative, the equivalent function `List[`$elem_1, \ldots, elem_n$`]` can be used to construct the list.

`Table[`$f, \{i, imin, imax\}$**`]`** builds a vector of length $imax - imin + 1$ by running the function f with sequential values $i = imin, \ldots, imax$.

The expression `Table[{ts[[i]]},{i,1,Length[ts]}]`, for instance, constructs a new list with each element of `ts` enclosed in an extra list. If `ts`, for example, is `{1,5,2}`, then the result will be `{{1},{5},{2}}`.

`Table[`$f, \{i, imax\}$**`]`** is similar to the version above, but with the function f evaluated for values $i = 1, \ldots, imax$.

The following operations allow investigating the structure of a list by measuring its dimensions:

- `Length[`$list$`]` returns the number of elements in the topmost level of a list. If the list is a nested structure, each sub list is counted like a single element. The length of the empty list `Length[{}]` is 0.
- `ArrayDepth[`$list$`]` determines the maximum depth of a possibly nested list structure. The *ArrayDepth* is 1 for a linear list and 0 if the argument is an unstructured object. Note, that `ArrayDepth[{}]` returns 1, because {} is a list, although it contains no elements.

Lists can be treated in an array-like manner, picking out or setting individual elements in the list by giving it's *index* or a list of indices enclosed in double square brackets ⟦ `and` ⟧. Instead of double square brackets, a sequence of two square brackets `[[` and `]]` can be used. The numbering of the elements of a list ranges from 1 to `Length[`$list$`]`. If a list is constructed in a form like `{1,2,3,4,5}`, the elements are numbered from left to right. The expression

v[[2]] for instance, returns the second element of v. The expression v[[{2, 4}]] returns a list of length 2 containing the elements on position 2 and 4 in the same order as in v.

Alternatively, list elements can be accessed in a back to front order with negative indices. v[[−1]] for instance, accesses the last element of list v. Both types of indices can be used together. For example: v[[{1, −1}]] returns a list containing the first and the last element of v. If v has only one element, both entries in the returned list contain this value. If the given index is higher or lower than the length of the list, *Mathematica* emits an error message.

Certain extractions are so important, that they have their own functions:

First[*list***]** returns the first element of a list. This function is identical with *list*`[[1]]`.

Last[*list***]** returns the last element of a list. The same result can be obtained with the expression *list*`[[-1]]` or *list*`[[Length[`*list*`]]]`.

Rest[*list***]** returns a copy of *list* where the first element is dropped.
The result of `Rest[{1,2,3,4}]`, for example, is the list `{2,3,4}`.

If a list contains a nested structure, multiple indices can be used to access elements on a deeper level. If the number of used indices is smaller then the depth of the structure, a substructure of the list is returned. The following example shows the handling of a matrix-structure:

`m={{11,12},{21,22}}` → $\begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$
`m[[1,2]]` → 12
`m[[1,1]]` → 11
`m[[1]]` → {11,12}

The function `Take[]` provides a way to get pieces of a list:

Take[*list*`,`*n*`]` returns the first n elements of *list*, if n is a positive integer. If n is a negative integer, the last n element of *list* are returned.

Take[*list*`,`{*m*, *n*}`]` returns a sequential sub list of elements from *list*, starting with the m'th element and ending with the n'th element. m and n can be positive or negative integers specifying the position of the first or last element in front to back and back to front order respectively.

Testing and Searching List Elements

MemberQ[*list*`,` *form*`]` tests, whether *form* is an element of *list* and returns either `True` or `False`.

FreeQ[*list*`,` *form*`]` tests, whether *form* is no element of *list* and returns either `True` or `False`.

Changing the Structure of a List

The structure of a nested list can be flattened in various ways:

`Flatten[`*list*`]`** removes all nested levels creating a linear list of all elements on the innermost levels.

Example: `Flatten[{{{1},{2}},{{3},{4}}}]` returns `{1,2,3,4}`.

`Flatten[`*list*`,`*n*`]`** limit's the degree of flattening by only removing n levels down from the top of the nested structure.

Example: The expression `Flatten[{{{1},{2}},{{3},{4}}},1]` returns the list `{{1},{2},{3},{4}}`.

Selecting Elements from Lists

The operation `Select[`*list*`,`*condition*`]` matches the elements of *list* against a certain *condition* and returns a new list including only the matching elements. The expression *list* can be any valid *Mathematica* expression representing a list. The expression representing *condition* must be a logical expression that evaluates either to `True` or `False`.

Select evaluates the logical expression *condition* for each element of *list*. Inside *condition* the actually examined element can be referenced with the symbol #[3], which acts as a placeholder representing the actual element. The placeholder # can be used repeatedly to reference to the actual processed element on several places inside the logical expression. The symbol & is used to mark the end of scope of the current # placeholder. This is necessary, because functions like *Select* iterating over the elements of a list can be used in nested structures.

Examples: The following piece of code defines a function, returning all nodes from a set V with a label matching the string vl:

```
gSelectNodeByLabel[V_, vl_]:= Select[V, gNodeLabel[#] == vl &]
```

The next excerpt from the coverage preservation notebook shows the multiple use of the # placeholder:

```
gSelectEdgeById[R_, eh_, et_] :=
    Select[R, (gEdgeHead[#] == eh) ∧ (gEdgeTail[#] == et) &]
```

The result of the `Select` function is a list, including all elements that match the given condition. If no element matches the condition, the empty list `{}` is returned.

Set Operations

The "natural" behaviour of a list is similar to a tuple. It provides a fixed order of elements and allows multiple occurrences of the same element. Some functions are especially designed to

[3]The symbol # stands for a *Mathematica* feature called *pure function*. Pure functions allow defining functions without given them explicit names. For more detailed information on pure functions, please refer to the *Mathematica* online manual.

treat a list in a set-like manner, removing duplicates of elements and sorting the elements in some standard order[4].

The following functions are designed to treat lists in a set-like manner. It doesn't matter, whether the arguments of the set-functions are already in set-style or not, the result of these operations will always be a set:

`Union[`$list_1, \ldots, list_n$**`]`** returns a list of the distinct elements of all given lists. Alternatively, the mathematical notation $list_1 \cup \ldots \cup list_n$ reveals the same result.

`Intersection[`$list_1, \ldots, list_n$**`]`** returns a list of the elements that are common to all given lists. The same result can be obtained using the set-theoretic notation $list_1 \cap \ldots \cap list_n$.

`Complement[`$list_1, \ldots, list_n$**`]`** returns a list of all elements that are in $list_1$ but not in any of the other lists.

`Union[`$list$**`]`** converts a list into a set by sorting the elements in an arbitrary manner and removing any duplicates.

Rearranging Lists

The functions described in this paragraph manipulate lists in a way that avoids changing the tuple-like behaviour of a list. Such functions do not remove multiple occurrences of elements and avoid unintentional changes in the order of elements.

`Append[`$list$**`,`**$element$**`]`** adds one $element$ at the end of $list$ and returns the new list.

`Prepend[`$list$**`,`**$element$**`]`** inserts one $element$ at the beginning of $list$ and returns the new list.

`Join[`$list_1$**`,...,`**$list_n$**`]`** concatenates $list_1, \ldots, list_n$ in the given order and returns the resulting list. In contrast to `Union[]`, the `Join[]` function does not remove multiple occurrences of elements and does no resorting of elements.

Applying Functions to Lists

A common task of list handling is to apply functions in various ways to the elements of a list. *Mathematica* provides several ways to do this:

`Map[`$function, list$**`]`** wraps $function$ around each element of the outermost level of $list$. In other words, it calls $function$ with each entry on the first level of $list$ as argument. `Map` returns a list with the results of expression $function$ for each element of $list$.

If only a function name is given for expression $function$, the function is called with each element of $list$ as the only one argument. More complex expressions can be created

[4]The standard behaviour of set-functions in *Mathematica* 5.2 is to bring elements into a numerically or lexically sorted order.

using the # symbol to reference the actually examined entry of $list$ (please refer to the description of *Select* above).

Example: The following excerpt from the code coverage notebook defines a function, which converts a list of node tuples into a list of node identifiers:

```
gDecisionIdSet[d_] := Map[gNodeId, gDecisionNodes[d]]
```

`gDecisionNodes` is a function, returning a list of nodes of decision `d`. `gNodeId` returns the identifier of the node, given as argument. `Map` applies `gNodeId` to each element of the list returned by `gDecisionNodes` and therefore extracts the identifier of each node of the decision. The result is returned as a list, which contains all identifiers.

`Apply[`$f, list$**`]`** applies f directly to the elements of a list, making each element of $list$ a separate argument of f.

Example: The expression `Apply[f,{a,b,c}]` gives `f[a,b,c]`.

`Outer[`$f, list_1, list_2, n$**`]`** is a generalized version of the Cartesian product of lists. It takes all possible combinations of elements of the lowest level from $list_1$ and $list_2$ calling the function f with each combination as arguments. The optional value n determines the level of sub lists that is treated as separated elements.

Example: The expression `Outer[f,{a,b,c},{1,2}]` gives the result `{{f[a,1], f[a,2]},{f[b,1], f[b,2]},{f[c,1],f[c,2]}}`.

A.4 Control Flow Expressions

Procedural programming normally involves some kind of execution flow control. Dependent on the result of evaluating a certain condition the procedure will follow different execution steps. Although *Mathematica* provides several different ways for execution flow control, only a few are used to implement the automatic code coverage analysis: `If`, `While` and `For`. The principle function of control flow expressions are similar to `if`, `while` and `for` control structures in programming languages like C/C++. But there are some important differences, not only in syntax but also in their semantic.

`If[`$test, then, else$**`]`** executes expression $then$, if the condition $test$ evaluates to `True`. If $test$ evaluates to `False`, expression $else$ is executed. If the result of evaluating $test$ is neither `True` nor `False`, both branches of the *If* will remain unevaluated[5].

`While[`$test, body$**`]`** evaluates $body$ repeatedly as long as $test$ evaluates to `True`.

`For[`$start, test, incr, body$**`]`** executes the expression $start$ and then repeatedly executes $body$ and $incr$, until evaluating expression $test$ fails to be `True`.

The loop-test is always executed before the body of the loop. If the loop test fails, the *While* and *For* loop terminates. Boolean expressions are evaluated in a definite sequence with shortcut

[5]The reason for this behaviour is *Mathematicas* ability to perform symbolic computations.

semantic. Evaluating a combination of Boolean expressions stops, if any of the tests reveals a final result. This behaviour allows sequences of tests, where later tests make sense only if the earlier ones are satisfied [58].

The result of a control flow expression is always the result of the last expression evaluated. The result of a compound expression is always the result of the last expression executed. If a compound expression is finished with a semicolon, or if an empty expression is executed as last operation however, the evaluation result will be *Null*.

A.5 Operations on Strings

This is a brief overview of some string functions. More information can be found in the *Mathematica* online manual.

`StringLength[`s**`]`** returns the number of characters in string s.

`StringJoin[`$\{s_1, \ldots, s_n\}$**`]`** concatenates several strings of a list together.

`StringTake[`$s, \{n\}$**`]`** takes the n^{th} character of string s.

`StringPosition[`$s, \{sub_1, \ldots, sub_n\}$**`]`** gives a list of the starting and end positions of any occurrences of substrings sub_i in the string s.

`ToLowerCase[`s**`]`** generates a string where all letters of s are lower case characters.

APPENDIX B

Mathematica Notebook Listing

This section provides an excerpt listing of the *Mathematica* implementation with algorithmic solutions and proofs for the coverage preservation analysis. The notebook was written for *Mathematica* version 5.2.

Automatic analysis can be started by running the analysis function

MatrixForm[cPresAnalysis[−1]]

Note, that the prefix *MatrixForm* is optional for presenting the output in a more appealing form. The argument -1 means "no intermediate output". Running the analysis function as described above will produce the following output, for example:

Use Case	SC w/o C.	SC w. C.	CC	DC	MCDC	PC
Useless Code Elimination	True	True	True	True	True	True
Full-Evaluated If-Then-Else	True	True	True	True	True	True
Shortcut If-Then-Else	True	True	False	True	False	False
Empty Else Shortcut AND	True	True	False	True	False	False
Empty Else Shortcut OR	True	False	False	True	False	False
Loop Peeling (k=1)	False	False	False	False	False	False
Loop Inversion	True	True	False	False	False	False
Loop Fusion	True	True	True	True	True	True
Loop Interchange	True	True	False	True	False	False
Loop Unrolling (k=2)	False	False	False	False	False	False
Strip Mining	True	True	False	True	False	False
Loop Tiling	True	True	False	True	False	False
Loop Unswitching	True	True	False	True	False	False
Software Pipelining	False	False	False	False	False	False
Branch Optimization	False	False	True	True	True	True

Coverage Preservation

Initializations and Globals

■ Loading Libs

Using *GraphPlot* for visualizing the program graphs in a (more or less) appealing form. Note, that the ability of version 5.2 to produce nice drawings is very restricted.

```
<< DiscreteMath`GraphPlot`;
```

■ Defining Globals

The following variables serve as global values to control program properties and program execution. Some of them are only used in one place, but they are given here to collect them all on a single place for easyer maintenance.

■ outputSet

outputSet is used to control the details of function-output prints. *outputSet* is a set variable. Each element represents a certain type of output that is displayed during running a function. The higher the value of the set element the more detailed is the output, e.g. *0* represents high level output, printing only what's going on without further details. In contrast, *9* represents output of internal information produced in deeper levels of the functions. An empty set produces no output. The *outputSet* can be changed at any time by assigning a new value to the variable *outputSet*.

```
outputSet = {0, 1};
```

■ loopDecisionKeyword

loopKeywords defines the keywords of decision tags to identify a decision as loop-decision.

```
loopDecisionKeywords = {"loop", "while"};
```


■ Decision True/False Set

To model decisions with tendency, we use a set of symbols to label the outcome within one decision. To restrict possible path combinations through a decision these symbols must be used on the outgoing edges of a decision too. On the external edges outgoing from a decision it is always necessary to present a final decision, because this is required when calculatiing IV_T or IV_F of a decision. Therefore it is necessary to map the used symbols to *True* or *False*. We do this by creating two global sets, which hold the symbols for *True* and *False*. For convenience we use the characters "0","1","2","3", "4" and "T" for *True* and "5",...,"9" together with "F" for *False*. If necessary, this set can be expanded using additional characters. The only restriction is, that the symbols "T" and "F" must be part of these sets and that the used symbol must not comprise more than one character, since the decision label is only one character long.

```
DecTrueFalseSet =
  {{"T", "0", "1", "2", "3", "4"}, {"F", "5", "6", "7", "8", "9"}};
```

■ Changing Built-in Properties

We often use names with small differences. *Mathematica* produces a bunch of warnings for that. Setting the following property to *off* will avoid these warning messages.

```
Off[General::spell]
```

Graph-Functions

This part contains *Mathematica*-versions of graph functions for *Analysis Control Flow Graphs* (aCFG). For convenience we use the abbreviation CFG also for Analysis CFGs. Without further notice, the abbreviation CFG addresses an aCFG. Because of the history of the framework decelopment, the aCFG is also named "Extended Control Flow Graph" (eCFG).

■ Definition "Extended Control Flow Graph" (CFG)

The following functions are helpers to extract the different sets defining the aCFG without concern about the exact position in the tuple or the actual implementation. To avoid confusions between these functions and *Mathematica*'s built in functions we use a notation with a preceeding *g* like for all general graph and control flow graph functions (*Note:* The *Mathematica*-Book defines, that all non standard functions should start with a lower case letter to distinguish them from built-in functions).

gB selects the set of basic blocks, g*C* returns the set of conditions nodes and g*D* returns the decision set. Finally *gV* returns the set B ⋃ C. *gR* gives the edge set of the graph *G* and finally *gST* returns the tuple ⟨*s*,*t*⟩ consisting of *start* and *termination* node.

```
gB[G_] := G[[1]]
```

```
gD[G_] := G[[2]]
```

```
gC[G_] := Flatten[Map[Rest, gD[G]], 1]
```

```
gV[G_] := gB[G] ⋃ gC[G]
```

```
gR[G_] := G[[-3]]
```

```
gST[G_] := {G[[-2]], G[[-1]]}
```

The following function will construct an aCFG from given sets *bB*, *dD*, *rR* and a tuple *stST* containing the start node as first and the termination node as last component. The arguments are used to construct the components *B*,*D*,*R*,*s*,*t* of an extended CFG.
Note, that this function is not used consequently in the use cases.

```
eCFG[bB_, dD_, rR_, stST_] := {bB, dD, rR, First[stST], Last[stST]}
```

■ Selection Functions for Nodes and Node-Components

The following functions will extract informations from a node tuple *v*. A node is a tuple ⟨λ,i,ri⟩ where λ is a (textual) *node tag* and *i* is a *unique node-id*. Additional, a node could carry a *related-id*. *ri*, which defines functional relationships between nodes. Detailed information on the functional node relations are given in the section about transformation relations. The component *ri* is used in case of transformed graphs to define the functional relationships between nodes of transformed and untransformed graphs. If there is no functional relationship, the component is missing or initialized with -1. Please note, that the node tag need not to be unique for all nodes, but the node id must be unique. *gNodeTag* returns the tag of a node, *gNodeId* returns the unique (internal) node identifier and *gNodeRelId* returns the related node id.

```
gNodeTag[v_] := v[[1]]
```

```
gNodeId[v_] := v[[2]]
```

```
gNodeRelId[v_] := If[Length[v] > 2, v[[3]], -1]
```

gSelectNodeById is a helper to examine an (arbitrary) node-set *V* for the presence of a node with a given node-id. *gSelectNodeByTag* searches for nodes with a given node-label. Note, that the function *gSelectNodeById* always returns a set of nodes, although a correct node-set of an aCFG should only contain one node with the given id.

```
gSelectNodeById[V_, iv_] := Select[V, gNodeId[#] == iv &]
```

```
gSelectNodeByTag[V_, vl_] := Select[V, gNodeTag[#] == vl &]
```


■ Selection Functions for Edge Components

These functions retrieve components of the edge structure. g*EdgeHead* returns the node-id., the edge is starting from, g*EdgeTail* returns the termination node-id of edge *e*. g*EdgeLabel* returns the string the edge is labelled with and *gEdgeValuation* returns the valuation value attached to *e*. *gEdgePathMarker* is a function to return a path-identifier. If the components 4 or 5 do not exist, an empty set is returned.

```
gEdgeHead[e_] := e[[1]]
```

```
gEdgeTail[e_] := e[[2]]
```

```
gEdgeLabel[e_] := e[[3]]
```

```
gEdgeValuation[e_] := e[[4]](*If[Length[e]≥4,e[[4]],{}]*)
```

```
gEdgePathMarker[e_] := If[Length[e] ≥ 5, e[[5]], {}]
```

gSelectEdgeById searches a given edge set R for the presence of edges starting at a node with id eh and terminating at a node with id et. The edges are returned in a set of edges.

```
gSelectEdgeById[R_, eh_, et_] :=
 Select[R, (gEdgeHead[#] == eh) ∧ (gEdgeTail[#] == et) &]
```

■ Selection Functions for Decision Components

A decision is implemented as a flat tuple $\langle \lambda, c_1, c_2, \ldots \rangle$ of arbitrary length, where λ is a decision tag and c_i are tagged nodes or related tagged nodes (like the nodes defined above) the decision is composed of. The selection function *gDecisionTag* returns the tag of decision *d*, *gDecisionNodes* returns the set of condition-nodes, the condition is composed of.

```
gDecisionTag[d_] := First[d]
```

```
gDecisionNodes[d_] := Rest[d]
```

The following functions extract sets of node-id's and related node-id's respectively of decision *d*:

```
gDecisionIdSet[d_] := Map[gNodeId, gDecisionNodes[d]]
```

```
gDecisionRelSet[d_] := Union[Map[gNodeRelId, gDecisionNodes[d]]]
```

gSelectDecisionById returns the set of decisions from the decision-set *ds*, that contains a node with the given *id* or which set of node-id's is identical to the given id-set *id*. The result is always a set of decisions, although in a correct CFG there should exist at most one matching decision.

```
gSelectDecisionById[ds_, id_] :=
 If[ArrayDepth[id] == 0,
  Select[ds, MemberQ[gDecisionIdSet[#], id] &],
  Select[ds, gDecisionIdSet[#] == id &]
 ]
```

gSelectDecisionByTag retrieves all decisions of decision set *ds* matching a given decision tag *dl*.

```
gSelectDecisionByTag[ds_, dl_] := Select[ds, gDecisionTag[#] == dl &]
```

■ Other Edge Functions

gOutEdges determines the outgoing edges directed from the node with id *iv* to other nodes, g*InEdges* determines the edges terminating in the node with id *iv*. g*InEdgesH* and g*OutEdgesH* do the same, but for hypernodes defined by a set *iv* of node-id's. g*EdgeSetH* returns the internal edges of the subgraph determined by the hypernode *v*.

```
gOutEdges[R_, iv_] := Select[R, gEdgeHead[#] == iv &]
```

```
gOutEdgesH[R_, iv_] :=
 Select[R, ¬ MemberQ[iv, gEdgeTail[#]] ∧ MemberQ[iv, gEdgeHead[#]] &]
```

```
gInEdges[R_, iv_] := Select[R, gEdgeTail[#] == iv &]
```

```
gInEdgesH[R_, iv_] :=
 Select[R, ¬ MemberQ[iv, gEdgeHead[#]] ∧ MemberQ[iv, gEdgeTail[#]] &]
```

```
gEdgeSetH[R_, iv_] :=
 Select[R, MemberQ[iv, gEdgeTail[#]] ∧ MemberQ[iv, gEdgeHead[#]] &]
```

gLabOutEdges and *gLabOutEdgesH* selects outgoing edges with a particular label *l* and returns them in a set. If an edge with the required label does not exist, it returns the empty set.

```
gLabOutEdges[R_, iv_, l_] :=
 Select[R, (gEdgeHead[#] == iv) ∧ (gEdgeLabel[#] == l) &]
```

```
gLabOutEdgesH[R_, iv_, l_] := Select[R,
  ¬ MemberQ[iv, gEdgeTail[#]] ∧ MemberQ[iv, gEdgeHead[#]] ∧ gEdgeLabel[#] == l &]
```

gSLabOutEdges is a selective version of *gLabOutEdges*. The user provides a string function sf which is applied to the label before comparing it with the string *l*. *gSLabOutEdgesH* provides the same function but for hypernodes. *gSLabOutEdgesHS* is a special version of the hypernode-function, which selects all outgoing edges where the label-component selected by *sf* is a member of a label set *ll*.


```
gSLabOutEdges[R_, iv_, sf_, l_] :=
 Select[R, (gEdgeHead[#] == iv) ∧ (sf[gEdgeLabel[#]] == l) &]
```

```
gSLabOutEdgesH[R_, iv_, sf_, l_] := Select[R, ¬ MemberQ[iv, gEdgeTail[#]] ∧
    MemberQ[iv, gEdgeHead[#]] ∧ sf[gEdgeLabel[#]] == l &]
```

```
gSLabOutEdgesHS[R_, iv_, sf_, ll_] :=
 Select[R, ¬ MemberQ[iv, gEdgeTail[#]] ∧ MemberQ[iv, gEdgeHead[#]] ∧
    MemberQ[ll, sf[gEdgeLabel[#]]] &]
```

gInDegree(H) and g*OutDegree(H)* calculates the number of incoming/outgoing edges of a node with index *iv* or a set *iv* of node-indices.

```
gOutDegree[R_, iv_] := Length[gOutEdges[R, iv]]
```

```
gInDegree[R_, iv_] := Length[gInEdges[R, iv]]
```

```
gOutDegreeH[R_, iv_] := Length[gOutEdgesH[R, iv]]
```

```
gInDegreeH[R_, iv_] := Length[gInEdgesH[R, iv]]
```

The following functions are simple helpers returning the predecessor edge set or the successor edge set sharing a common endpoint with a given edge.

```
gPredEdges[R_, e_] := gInEdges[R, gEdgeHead[e]]
```

```
gSuccEdges[R_, e_] := gOutEdges[R, gEdgeTail[e]]
```

■ Condition-, Decision- and Relation-Label

To allow connecting the outcome of conditions to an arbitrary outcome of a decision we use a two-parted condition-/decision-label on edges outgoing from a decision node. The first letter defines the outcome of the current condition while the second letter defines the decision outcome. For the condition- and the decision-part the letters "T" and "F" are used for *true* and *false* outcome of the condition/decision. Only for the decision-part the letter "X" is used if the outcome of the decision is not decided when passing the current edge. Clearly "X" is only allowed inside a decision hypernode. All edges leaving a decision-hypernode must have one of the decision labels "T" or "F". Note, that some additional characters can be used for decision outcome.

The functions *gCLabel* and *gDLabel* extract the **C**ondition part or **D**ecision part of an edge-label. The return-value is a one character string. If the label is the empty string both functions return the empty string. If the label only consists of the condition-label the *gDLabel* function returns the empty string. The *gRLabel*-function accesses the *functional relationship* character. If the character is not present, the default character "=" is returned.

```
gCLabel[l_] := If[StringLength[l] > 0, StringTake[l, {1}], ""]
```

```
gDLabel[l_] := If[StringLength[l] > 1, StringTake[l, {2}], ""]
```

```
gRLabel[l_] := If[StringLength[l] > 2, StringTake[l, {3}], "="]
```

■ gLabelJoin

...joins edge labels given in the set *ls*. If the set includes condition labels of both outcomes (*True* and *False*) the resulting condition label is *V*, else the resulting label is *True* or *False* according to the values in the set. If the decision labels are all members of the same outcome set, the decision label is set to the matching outcome *True* or *False*. Otherwise or if one of the multiple labels contains *X*, the resulting decision component of the unified label is set to *X*. The functional relation labels are omitted.

```
gLabelJoin[ls_] := Module[{cl, dl, c, d},
  (* Divide label into condition/decision components.
   *)
  cl = Union[Map[gCLabel, ls]];
  dl = Union[Map[gDLabel, ls]];

  (* The condition label can only contain "T" or "F". Therefore
    a length of 2 means, that both are member in "cl".
   *)
  If[Length[cl] > 1, c = "V", c = First[cl]];

  (* Check the decision symbols if they are all from one decision
    set and if so, select the leading decision symbol. In all
    other cases, the result must be "X", because the decision
    symbols comprise "X" or symbols from different True/False
    decision symbol sets.
   *)
  If[Complement[dl, DecTrueFalseSet[[1]]] == {},
   d = First[DecTrueFalseSet[[1]]],
   (* else *)
   d = If[Complement[dl, DecTrueFalseSet[[2]]] == {},
     First[DecTrueFalseSet[[2]]], "X"]
  ];

  c <> d
 ]
```


■ Path Functions

■ gPathSet

The function *gPathSet* examines an edge set *R* for the existence of paths starting at some edge from the edge set *ss* and ending at some edge of the edge set *ts*. The returned set contains all possible combinations of edge-sequences from one of the edges of *ss* to one of the end edges of *ts*, including the corresponding start-edge and end-edge from the input set. The returned paths all contain single lines of edges without forks and joins. It is not guaranteed, that each of the given start-edges and end-edges are part of some path in the result set. If there are any loops the result set contains one path without the loop (immediate exit) and one path where the loop is executed exactly once.

The implementation of the function builds up the paths from the end to the beginning. It maintains two sets: *explore* for unfinished paths and *pathList* for finished paths (that means that these paths contain one of the given start edges). *explore* initially holds a path-set, where each path only includes one of the input end-edges. Inside a loop, an arbitrary path *p* of the *explore* set is taken (and removed from *explore*), expanded with the predecessor edge of the first path-edge and re-unified with *explore*. If the first edge of the path has more than one predecessor, then a certain number of pathes is produced and re-unified with *explore*. If there are more than one possibilities to expand a path, only the expansion which do not add an edge, which is already in the path, is processed. This avoids endless loops and on the other hand produces pathes, that walk through a loop exactly one time. If a path reaches the start-node without passing the given start-edges, it is earased. At the end of the path expansion phase, all pathes which include one of the starting-edges are moved from *explore* to *pathList*. The loop ends if nothing's left to *explore*.

```
gPathSet[R_, ss_, ts_] :=
 Module[{i, explore, pred, expandedPath, pathList = {}, p},
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ Path-set from ", MatrixForm[ss], " to ", MatrixForm[ts]]
  ];

  (* Construct the initial path-list by adding each single edge in
     "ts" to explore. Check if there are already finished pathes
     and move them to "pathList".
   *)
  explore = Table[{ts[[i]]}, {i, 1, Length[ts]}];
  pathList = pathList ⋃ Select[explore, (# ⋂ ss) ≠ {} &];
  explore = Complement[explore, pathList];
  If[MemberQ[outputSet, 9],
   Print["⟨9⟩ (gPathSet) Init. → pathList: ", pathList];
   Print["⟨9⟩ (gPathSet)          explore: ", explore]
  ];

  While[(explore ≠ {}),
   (* Take and remove an arbitrary path from explore (we use the
        first in the list).
    *)
   p = First[explore];
   explore = Rest[explore];

   (* Expand the path by taking the predecessor of the first
      element of the selected path prepending the new edge.
      If there are no predecessors (the start node of the graph
        is reached without passing the given start-edges)
      discard the path. If there are more than one predecessors
      expand only if the edge is not already in the path to avoid
```

```
    endless cycles.
   *)
  pred = gPredEdges[R, First[p]];
  If[MemberQ[outputSet, 9],
   Print["〈9〉 (gPathSet) Iter. → pred. of ", First[p], ": ", pred];
  ];

  If[pred ≠ {},
   If[Length[pred] > 1,
    pred = Select[pred, ¬ MemberQ[p, #] &];
    (*pred=Select[pred,¬gDoubleCirclePath[#,p]&];*)
    If[MemberQ[outputSet, 9],
     Print["〈9〉 (gPathSet) Loop cleaned pred.: ", pred];
    ]
   ];

   explore = explore ⋃ Table[Prepend[p, pred[[i]]], {i, 1, Length[pred]}];

   (* Move finished pathes to "pathList" and remove it from "explore".
    *)
   pathList = pathList ⋃ Select[explore, (# ⋂ ss) ≠ {} &];
   explore = Complement[explore, pathList]
  ]; (* If pred ≠ {} *)

  If[MemberQ[outputSet, 9],
   Print["〈9〉 (gPathSet) Iter. → pathList: ", pathList];
   Print["〈9〉 (gPathSet)          explore: ", explore]
  ]

 ]; (* While *)

 pathList
]
```

■ gShortestSubPathSet

Sometimes it is necessary to select a subpath from a path *p* which only consist of edges originating a certain vertices-set *g*. This is done by the function *gShortestSubPathSet*, which extracts the shortest subsequence of edges from a path *p* that meet the condition, that (1) each subsequence selected contains only edges originating at nodes from one vertices-set of *gG* and (2) the nodes of this set are included in the path at most once but (3) as most as possible. Condition (2) means, that any second occurence of an element of a vertices-set in an edge-sequence closes the current subpath and opens a new one. Condition (3) means that a subpath is not complete as long as there exists a subsequent edge originating at a node from the current vertices-set which is not already head-node of the edges already included in the subpath.


```
gShortestSubPathSet[p_, gG_] :=
 Module[{flatG, explore, subPath, expDec, pathList = {}},
  explore = p;
  flatG = Flatten[gG];
  (*Print["gG: ",gG,"     flatG: ",flatG];*)

  While[explore ≠ {},
   (*Print["#: ",Length[explore],"   explore: ",explore];*)

   If[FreeQ[flatG, gEdgeHead[First[explore]]],
    explore = Rest[explore],

    (* else *)
    expDec =
     Complement[First[Select[gG, MemberQ[#, gEdgeHead[First[explore]]] &]],
      {gEdgeHead[First[explore]]}];
    (*Print[First[explore]," starts sub-path, remaining
        subset to explore: ",expDec];*)

    subPath = {First[explore]};
    explore = Rest[explore];

    While[(explore ≠ {}) ∧ (MemberQ[expDec, gEdgeHead[First[explore]]]),
     subPath = Append[subPath, First[explore]];
     expDec = Complement[expDec, {gEdgeHead[First[explore]]}];
     explore = Rest[explore];
    ];

    pathList = Append[pathList, subPath];
    (*Print["add ",subPath," to path-list. "];*)

   ] (* While *)

  ]; (* While *)

  pathList
 ]
```

■ Labelled Path

To get pathes related to certain condition/decision-labels, we need a helper function to check, wether or not a path *p* fits a certain edge-label pattern. A path fits a certain edge-label pattern Λ, if the labels of all edges in the path are elements of the set Λ. The function *gSLabelPath* checks a path *p*, if it fits an edge-label pattern *elp*. Like *gSLabOutEdges* the function can be supplied with the name of a filter-function to convert the edge label before evaluate them with respect to *elp*. The advantage of this method is, that structured labels can be processed without mention the values of the other labels (e.g. Decision labels can be processed without knowing the values of condition labels).

The function returns *True* if the path fits the label pattern, *False* otherwise. The empty path always returns *True*.

```
gSLabelPath[p_, elp_, f_] :=
 Apply[And, Map[MemberQ[elp, f[gEdgeLabel[#]]] &, p]]
```

■ Breadth/Depth First Search Order

In some cases it is necessary or preferred to traverse a graph in a "top to bottom" order. Therefore the next functions provide a list, where the nodes of a transformation-graph are ordered in a sequence which represents their position relative to the start-node. Two different kinds of search orders are provided. *Breadth First Search* order places first all nodes with the same distance from the start-node before storing the nodes on the next level. In the *Depth First Search* order nodes are stored with increasing distances from the start node first before nodes on the same level are placed.

g*DfsOrderList* creates a list in *depth first search* order, g*BfsOrderList* creates a list of nodes in *breadth first search* order. The purpose of this functions is to create a list of node id's to walk through an aCFG in a "top to bottom" order. To avoid problems with any kind of incorrect graphs, the function finally adds all orphan nodes (if any exists) at the end of the list to make sure, that all nodes are in the list. The result list always starts with the start-node and does not contain the termination-node.

The function takes a set *V* of inner nodes, the edge set *R* and a tuple *ST=⟨ s,t ⟩* of start and termination node as argument. This argument-split gives the possibility to use this function for CFG's as well as for transformation graphs.

```
gBfsOrderList[V_, R_, ST_] :=
 Module[{wtl = {}, explore = {gNodeId[ST[[1]]]}, t = gNodeId[ST[[2]]], n},
  (* continue until nothing' s left to explore...
   *)
  While[explore ≠ {},
   n = First[explore];
   wtl = Append[wtl, n];

   (* when updating "explore" avoid to include the termination node or
      any node already processed or in process The new nodes are added
      at the end of the list to keep the bfs order.
    *)
   explore = Join[Rest[explore],
     Complement[Map[gEdgeTail, gOutEdges[R, n]], wtl⋃explore⋃{t}]];
  ];

  (* Complete the list by adding all nodes not in "wtl" at end of the list
    (except termination node).
   *)
  wtl = Join[wtl, Complement[Map[gNodeId, V], wtl]];

  wtl
 ]
```



```
gDfsOrderList[V_, R_, ST_] :=
 Module[{wtl = {}, explore = {gNodeId[ST[[1]]]}, t = gNodeId[ST[[2]]], n},

  While[explore ≠ {},
   (* continue until nothing' s left to explore...
    *)
   n = First[explore];
   wtl = Append[wtl, n];

   (* when updating "explore" avoid to include the termination node or
      any node already processed or in process The new nodes are added
      in front of the list to keep the dfs order.
    *)
   explore = Join[Complement[
      Map[gEdgeTail, gOutEdges[R, n]], wtl⋃explore⋃{t}], Rest[explore]]
  ];

  (* Complete the list by adding all nodes not in "wtl" at end of the list
    (except termination node).
   *)
  wtl = Join[wtl, Complement[Map[gNodeId, V], wtl]];

  wtl
 ]
```

■ Graph-Plot Functions

These functions support pictoral drawing of control flow graphs, using some built in *Mathematica* functions.

■ Support Functions for Graph-Plots

gPlotListEdge converts the edge given as input to the format required by *Mathematica*'s built in drawing function. This function is defined in two versions: The first one receives as input an edge and outputs a rule-list entry based on the node-id's. The second version receives a set of vertices together with the edge and returns a rule-list entry based on the node-labels.

```
gPlotListEdge[e_] := gEdgeHead[e] → gEdgeTail[e]
```

```
gPlotListEdge[e_, V_] :=
 gNodeTag[First[Select[V, gNodeId[#] == gEdgeHead[e] &]]] →
  gNodeTag[First[Select[V, gNodeId[#] == gEdgeTail[e] &]]]
```

gEdgePlotList converts the list based edge representation to a rule list for use with *Mathematica*'s drawing functions. If only an edge-set is given as argument, the nodes are named with their id's. If the node set is given in addition, the labels of the nodes are used to name the vertices in the plot-list.

```
gEdgePlotList[R_] := Map[gPlotListEdge, R]
```

```
gEdgePlotList[R_, V_] := Map[gPlotListEdge[#, V] &, R]
```

■ Plot-Functions

Now we are ready to draw a programs graph. We need two drawing functions, to draw the vertices and the edges of the graph.

gVertexDraw is the drawing function for vertexes. *labelDraw* set to *True* selects output of vertex-labels. If *idDraw* is set to True, id's are drawn. If both are *True*, the full node information is drawn, if both are *False*, only a point is drawn.

```
gVertexDraw[P_, pos_, id_, labelDraw_, idDraw_] :=
 Module[{theNode, labelColor},
  theNode = First[gSelectNodeById[gB[P] ⋃ gC[P] ⋃ gST[P], id]];
  labelColor = Black;
  If[MemberQ[gST[P], theNode],
   If[theNode == gST[P][[1]],
    labelColor = DarkGreen,
    labelColor = Red
   ]
  ];
  If[MemberQ[gC[P], theNode],
   labelColor = Blue
  ];

  If[labelDraw ∧ idDraw,
   {labelColor, Text[ToString[theNode], pos,
     Background → White, TextStyle → {FontWeight → "Bold"}]},
   (*else*)
   If[labelDraw,
    {labelColor, Text[gNodeTag[theNode], pos,
      Background → White, TextStyle → {FontWeight → "Bold"}]},
    (*else*)
    If[idDraw,
     {labelColor, Text[ToString[gNodeId[theNode]], pos,
       Background → White, TextStyle → {FontWeight → "Bold"}]},
     (*else*)
     {labelColor, Disk[pos, 0.02]}
    ]
   ]
  ]
 ]
```

gEdgeDraw is the drawing function for the edges.


```
gEdgeDraw[P_, e1_, e2_, pos1_, pos2_] :=
 Module[{arrowPos, textPos, theEdges, theLabels},
  textPos = {0.5 * pos1[[1]] + 0.5 * pos2[[1]], 0.5 * pos1[[2]] + 0.5 * pos2[[2]]};
  arrowPos = {0.25 * pos1[[1]] + 0.75 * pos2[[1]], 0.25 * pos1[[2]] + 0.75 * pos2[[2]]};
  theEdges = gSelectEdgeById[gR[P], e1, e2];
  theLabels =
   Prepend[Map[If[gEdgeLabel[#] ≠ "", StringJoin[":", gEdgeLabel[#]]] &,
     Rest[theEdges]], gEdgeLabel[First[theEdges]]];

  If[theLabels == {},
   {AbsoluteThickness[2],
    Arrow[{pos1, arrowPos}, HeadCenter → 0.8], Line[{arrowPos, pos2}]},
   {AbsoluteThickness[2], Arrow[{pos1, arrowPos}, HeadCenter → 0.8],
    Line[{arrowPos, pos2}], Blue,
    Text[StringJoin[theLabels], textPos, Background → White]}
  ]
 ]
```

gDrawCfg creates a plot of a complete Cfg *P*. In the extended version, *drawVertexLabels* and *drawVertexId* determine, if drawing of labels or drawing of node-id's should be performed. If both values are *False*, only points are drawn. If *drawVertexLabels* and *drawVertexId* are omitted, vertex-id's are drawn.

```
gDrawCfg[P_] := gDrawCfg[P, False, True]
```

```
gDrawCfg[P_, drawVertexLabels_, drawVertexId_] := Module[{gp, vl, vc},
  (* Mathematicas graph-functions introduce their own node id-scheme,
     therefore we need a translation list "vl" to map them to the
     node-id's originally used. "vc" holds a list of appropriate
     node-coordinates.
   *)
  gp = gEdgePlotList[gR[P]];
  vl = VertexList[gp];
  vc = GraphCoordinates[gp, Method → "SpringModel", PlotRange → Automatic];

  GraphPlot[gp, EdgeStyleFunction →
    (gEdgeDraw[P, vl[[#1]], vl[[#2]], vc[[#1]], vc[[#2]]] &), VertexStyleFunction →
    (gVertexDraw[P, vc[[#]], vl[[#]], drawVertexLabels, drawVertexId] &)]
 ]
```

Valuation-Functions

Now defining functions to extract the valuation information from the edges and prepare it for evaluation. For simplicity we only determine the single valuations and the union of all valuations. We do not deal with different combinations of unions, if a combination of more then two edges on incoming and outgoing side are present.

■ Determining Valuation Information

vOutValuations and v*InValuations* extracts the valuation-information of all outgoing/incoming edges of a node with index *iv* and returns it as a plain-set. The functions automatically determine, if *iv* is a single index or a set of node-indices. In cases where *iv* is a set only the incoming edges with source outside or the outgoing edges with destination outside the node-set are mentioned while edges between the nodes are ignored.

The returned set is a *flat valuation set*, meaning that the function returns the valuations in the form $\{\delta_i, \delta_k, \ldots\}$ where δ_i, δ_k etc. are the valuation set values taken from the edges. This is identical to the union of all valuations on the incoming/outgoing side of the node with id *iv*.

```
vOutValuations[R_, iv_] := Module[{oe},
  If[ArrayDepth[iv] == 0,
   oe = gOutEdges[R, iv],
   oe = gOutEdgesH[R, iv]
  ];
  If[oe ≠ {},
   Union[Flatten[Map[{gEdgeValuation[#]} &, oe]]],
   (* else *)
   {}
  ]
 ]
```

```
vInValuations[R_, iv_] := Module[{ie},
  If[ArrayDepth[iv] == 0,
   ie = gInEdges[R, iv],
   ie = gInEdgesH[R, iv]
  ];
  If[ie ≠ {},
   Union[Flatten[Map[{gEdgeValuation[#]} &, ie]]],
   (* else *)
   {}
  ]
 ]
```

■ Reachability/Satisfyability Valuation

The following functions are implementations for the IV_R, IV_T and IV_F functions. The results are coded in a way that allows evaluation of the valuation-flow relations. For convinience we do not use the subscript-notation. Additional we use a generalized version of the satisfyability relations IV_T and IV_F called IVS. This functions takes the required label as a third parameter.

The functions are implemented in a two stage manner. *cRIE* and *cSOE* first extract the edges with the valuation-sets relevant for IVR and IVS. In a second step, *IVR* and *IVS* extracts the valuation sets from the edges and returnes them in a flat set.

■ Reachability Input Edge and Reachability Valuation (IV_R)

Reachability **I**nput **E**dges (RIE) returns all incoming edges of node with id *iv*. The function automatically detects, if *iv* is a single node or a cluster to collect the input edges of a single node or the input edges to a hypernode. The result of RIE is corresponding to IVR except that IVR directly returns the valuation values.

```
cRIE[R_, iv_] := If[ArrayDepth[iv] == 0, gInEdges[R, iv], gInEdgesH[R, iv]]
```

cIVR is the implementation of $IV_R(x)$**.**

```
cIVR[R_, iv_] := Union[Map[gEdgeValuation, cRIE[R, iv]]]
```


■ Satisfyability Valuation (IV_T and IV_F)

vDecisionLabelSet returns a set with all decision labels used in the given set of edges *p* (application note: *p* is usually a path through a decision).

```
vDecisionLabelSet[p_] := Map[gDLabel[gEdgeLabel[#]] &, p]
```

Satisfyability **O**utgoing (or **O**utcome) **D**egree (SOD) returns the number of different outcomes of a node with node-index *iv* or a decision with a set *iv* of condition-id's. The function is a helper for valuation-relation handling.

```
cSOD[R_, iv_] :=
 Length[
  Union[
   If[ArrayDepth[iv] == 0,
    Map[gCLabel[gEdgeLabel[#]] &, gOutEdges[R, iv]],
    Select[DecTrueFalseSet,
     # ⋂ vDecisionLabelSet[gOutEdgesH[R, iv]] ≠ {} &]
   ]
  ]
 ]
```

Satisfyability **O**utput **E**dges (SOE) returns all outgoing edges of node *iv* marked with condition-label or decision label *l*. The function automatically detects, if *iv* is a single node or a hyper-node to read the correct label with one of the functions *CLabel* or *DLabel*. The result of SOE is corresponding to IVS with the difference, that IVS directly returns the valuation values.

```
cSOE[R_, iv_, l_] := If[ArrayDepth[iv] == 0,
  gSLabOutEdges[R, iv, gCLabel, l],
  gSLabOutEdgesHS[R, iv, gDLabel,
   Flatten[Select[DecTrueFalseSet, MemberQ[#, l] &]]]
 ]
```

IVS is a general implementation for $IV_T(x)$ and $IV_F(x)$**.** The required satisfyability-label is given as Parameter.

```
cIVS[R_, iv_, l_] := Union[Map[gEdgeValuation, cSOE[R, iv, l]]]
```

■ Documentary Functions

The functions of this section are for documetary purposes.

vUnionRelFormat prepares print of valuation-set union *vu* , which is internal stored as $\{ \delta_i, \delta_j, \ldots \}$, by expanding the list to form $\delta_i, \bigcup, \delta_j, \bigcup, \ldots$

```
vUnionPrefix[n_] := {"⋃", n}
```

```
vUnionRelFormat[rg_] :=
 Prepend[Flatten[Map[vUnionPrefix, Rest[rg]]], First[rg]]
```

vRelationPrint is a extended printing function for valuation relations. The purpose of this function is to print relations from a relation set in a more readable form like $\delta_i \supseteq \delta_j$. The expression is given in a three parted form where *lhs* defines the expression on the left hand side, *rc* the relation character and *rhs* is the expression on the right handed side.

```
vRelationPrint[ps_, lhs_, rc_, rhs_] :=
 Print[ps, MatrixForm[vUnionRelFormat[lhs], TableDirections → {Row},
   TableSpacing → {0.1}], " ", rc, " ", MatrixForm[
   vUnionRelFormat[rhs], TableDirections → {Row}, TableSpacing → {0.1}]]
```

vPrintRelationSet is a printing function for a complete relation set. It receives the relation graph *rg* as an input argument and prints the relations in form of ⊇ and == relations, that means that this function checks for each relation $\delta \supseteq \rho$ if also $\rho \supseteq \delta$ is in the relation and dependent on that is decides to print ⊇ or ==. The prefix string *ps* is printed in front of each line.

```
vPrintRelationSet[rg_, ps_] := Module[{explore = rg, r1, r2},
  While[explore ≠ {},
   {r1, r2} = First[explore];
   If[MemberQ[explore, {r2, r1}],
    vRelationPrint[ps, r1, "==", r2],
    (* else *)
    vRelationPrint[ps, r1, "⊇", r2]
   ]; (* If *)
   explore = Complement[explore, {{r1, r2}, {r2, r1}}]
  ] (* While *)
 ]
```

vCompareRelSets compares two relation sets *set1* and *set2* and prints a list with remarks, which relation is member of which set.

```
vCompareRelSets[set1_, set2_] := Module[{inter, set1only, set2only},
  inter = set1 ⋂ set2;
  set1only = Complement[set1, set2];
  set2only = Complement[set2, set1];

  Print["⇒ ", Length[set1only], " of ", Length[set1],
   " elements only in SET 1.    ⇒ ", Length[set2only],
   " of ", Length[set2], " elements only in SET 2.    ⇒ ",
   Length[inter], " elements in common."];
  vPrintRelationSet[set1only, "  only Set1 → "];
  vPrintRelationSet[inter, "Set1 & Set2 → "];
  vPrintRelationSet[set2only, "  only Set2 → "]
 ]
```

■ Resolving Valuation Relations

The functions of this sections are helpers for the function v*ValRelGraph*. The function v*ValRelGraph* is the key element for calculating the valuation relations. It constructs a relation graph for a given aCFG *P* which is used to solve the relations between valuations. The nodes of this graph are the valuation sets (as defined before with the *vValRelObjects* function). A directed edge (δ_i, δ_j) is added to the graph, if $\delta_i \supseteq \delta$j. In case of δ_i==δj the graph contains edges between δ_i and δ_j in both directions.

vRelEdgeSet constructs the bundle of relations between the single valuation element *vSuper* and the elements in the set of valuations *vSubSet*, returning a set of tuples $\langle \delta, \rho \rangle$ with valuations δ=*vSuper* and $\rho \in$ *vSubSet* for any ρ in *vSubSet*.

```
vRelEdgeSet[vSuper_, vSubSet_] :=
 Flatten[Outer[{#1, #2} &, {vSuper}, vSubSet, 1], 1]
```

Next we define a helper function *vResolveValRel* to resolve all forward relations between the union of input-valuations *inSet* and the union of output valuations *outSet*. The function only determines the forward relations. If forward and backward relations are required, one must call this function a second time with swapped valuation sets.

```
vResolveValRel[inSet_, outSet_] :=
 {{inSet, outSet}} ⋃ Map[{inSet, #} &, Partition[outSet, 1]]
```

The function vResolveValEquality constructs all valuation relation between a incoming set of valuations *iv* and a outgoind set of valuations *ov* being in =-relation. The relations are resolved in forward direction as well as in backward direction. Additional it checks the input sets for validity and returns the empty set if at least one input set is empty. The main purpose of creating this function is allowing the use of the built-in *Mathematica*-function *Map* in *vValRelGraph* to iterate through node-set *B*.

```
vResolveValEquality[iv_, ov_] :=
 If[(iv ≠ {}) ∧ (ov ≠ {}), vResolveValRel[iv, ov] ⋃ vResolveValRel[ov, iv], {}]
```

vResolveNodeValuation calculates the valuation relation equality local to the input-valuation of node *in* and the output valuation of node *out*. The division *in*/*out* was made for using the function to determine relations between start- and termination-node. Calling the function with the same node for *in* and *out* determines the local relation inside one node (→ conservation axiom). *vResolveDecisionValuation* determines valuation relations of a decision, taking into account the type of the decision (loop decision or single branch-decision). The decision-type is determined by the label: if the decision label contains one of the keywords defined in the global *loopDecisionKeywords* (e.g.: "loop", "while"), the decision is treated as a loop, meaning that the outgoing *False* decision branch valuation is set equal to the decisions incoming valuation and the *True* branch is set to be the subset of the incoming valuation. In all other cases, the decision is treated "normal" with both exits set to be a subset of the incoming valuation.

```
vResolveNodeValuation[R_, in_, out_] := vResolveValEquality[
  vInValuations[R, gNodeId[in]], vOutValuations[R, gNodeId[out]]]
```

```
vResolveDecisionValuation[R_, decision_] :=
 Module[{dLabel = gDecisionTag[decision],
   dSet = gDecisionIdSet[decision], ivr, ivt, ivf, vr},
  ivr = cIVR[R, dSet];
  ivt = cIVS[R, dSet, "T"];
  ivf = cIVS[R, dSet, "F"];
  vr = vResolveValEquality[ivr, ivt ⋃ ivf] ⋃
    vResolveValRel[ivr, ivt] ⋃ vResolveValRel[ivr, ivf] ⋃
    If[StringPosition[ToLowerCase[dLabel], loopDecisionKeywords] ≠ {},
     vResolveValEquality[ivr, ivf], {}];

  vr
 ]
```

■ Simple Valuation Relation-Graph

vValRelGraph iterates through all elements of node-set *V* adding local relations according to the conservation axion. The undlying relation is given by *R*. Start- and termination-nodes are given in the tuple *st* where the first element is alway the start node and the second is always the termination node.

```
vValRelGraph[P_] :=
 Flatten[Map[vResolveNodeValuation[gR[P], #, #] &, gB[P] ⋃ gC[P]], 1] ⋃
  vResolveNodeValuation[gR[P], gST[P][[2]], gST[P][[1]]] ⋃
  Flatten[Map[vResolveDecisionValuation[gR[P], #] &, gD[P]], 1]
```

The valuation relation graph only contains "obvious" $\supseteq$-relations. Furthermore, there exists a whole bunch of more relations which can not be found directly by inspecting the control flow graph node-by-node. These additional relations must be investigated by transitivity which correspond to inspecting the nodes along a path. The following function determines the set of all valuation-sets being in a $\subseteq$-relation to a given node *v* defined by relation-set *R* (or in other words: it determines all nodes *w* where a path from *v* to *w* exists). We do this by inspecting each outgoing edge starting with *v* and adding the target-nodes to the relation node list. This procedure is repeated with all founded nodes until no uninspected reachable node is left.

■ Path Based Valuation Relation-Graph

■ sameEdge

Helper function, returns *True* if edge *e1* and edge *e2* are eqivalent in the first three components: head, tail and label (but not valuation set or path marker).

```
sameEdge[e1_, e2_] := (gEdgeHead[e1] == gEdgeHead[e2]) ∧
  (gEdgeTail[e1] == gEdgeTail[e2]) ∧ (gEdgeLabel[e1] == gEdgeLabel[e2])
```

■ vAddPathMarker

... adds path marker *pm* to the path-marker set of edge *e*, if *e* is part of path *p* and returns a modified version of *e*. If the edge is not included in path *p*, *e* is returned unmodified.

```
vAddPathMarker[p_, e_, pm_] :=
 If[Select[p, sameEdge[#, e] &] ≠ {},
  {gEdgeHead[e], gEdgeTail[e], gEdgeLabel[e],
   gEdgeValuation[e], gEdgePathMarker[e] ⋃ {pm}},
  e
 ]
```

■ vValidDecisionPath

A path through a decision is called a *valid decision path* if the decision labels of all edges in the path are markers for the same decision outcome, e.g. a decision path is valid if the decision labels of the edges carry "X" and "T" symbols, but it is not valid if one of the edges has decision label "F" and another edge on the path has decision label "T".

The function *vValidDecisionPath* checks if the given (sub-)path *p* is a valid decision path. This is done in the following way: First the decision labels of the edges in the given path are extracted and putted into a set. After removing the neutral symbols given by argument *ns* the remaining set must not consist of more than one symbol, otherwise the decision path is not valid.

```
vValidDecisionPath[p_, ns_] :=
 Length[Complement[vDecisionLabelSet[p], ns]] ≤ 1
```

■ vValidExecutionPath

Checks if path *p* is a possible execution path in an aCFG with respect to decisions-set *dD*. A path through an aCFG is a valid execution path if it comprises only valid decision sub-pathes for all its decisions.

```
vValidExecutionPath[p_, dD_] :=
 Apply[And, Map[vValidDecisionPath[#, {"X"}] &, gShortestSubPathSet[p, dD]]]
```

■ vPathTrace

Traces all pathes in an aCFG defined by execution relation *R* and start-/termination-node *st* using the path symbol *ps*. It returns the modified execution relation *R* with edges that contain an appended set-component with elements of the form ps_i where *ps* is the path symbol from the argument list and $i \in \mathbb{N}$ is a continual number, one for each path. If an edge is not part of any path between start-node and termination-node, it contains an empty set as last component.

```
vPathTrace[R_, st_, dD_, ps_] := Module[{i, r2, pP},
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ Tracing pathes through CFG ..."]
  ];

  pP = Select[gPathSet[R, gOutEdges[R, gNodeId[st[[1]]]],
     gInEdges[R, gNodeId[st[[2]]]]], vValidExecutionPath[#, dD] &];
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ Found ", Length[pP],
    " executable pathes, mark CFG-Edges with path-identifier ", ps]
  ];

  r2 = Map[{gEdgeHead[#], gEdgeTail[#],
      gEdgeLabel[#], gEdgeValuation[#], {}} &, R];
  If[MemberQ[outputSet, 9],
   Print["⟨9⟩ (vPathTrace) Initial execution-relation: ", MatrixForm[r2]]
  ];

  For[i = 1, i ≤ Length[pP], i++,
    r2 = Map[vAddPathMarker[pP[[i]], #, psi] &, r2];

    If[MemberQ[outputSet, 6],
     Print["⟨6⟩ Using ", psi, " for path ", pP[[i]]];
    ];
    If[MemberQ[outputSet, 9],
     Print["⟨9⟩ (vPathTrace) Intermediate relation ⇒ ", MatrixForm[r2]]
    ]

   ] (* For *)

   If[MemberQ[outputSet, 5],
    Print["⟨5⟩ Final relation: ", MatrixForm[r2]]
   ];

  r2
 ] (* Module *)
```

■ vResolvePathEquality

vResolvePathEquality resolves a valuation relation induced by a path relations between the (incoming) edge-set *e1* and the (outgoing) edge-set *e2*. The function, as a prerequisite, depends on the presence of path markers in both edges. Otherwise if at least on of the edges contains no path marker, the function always returns an empty relation.

Remark: although usually *e1* and *e2* will be sets of edges on corresponding incoming/outgoing sides, the function does not depend on the fact, that the edges share a comon endpoint nor does it depend on any preferred direction of the edges. Therefore the function can be used for any combination of edges.


```
vResolvePathEquality[e1_, e2_] := Module[{m1, m2, v1, v2, res},

  m1 = Union[Flatten[Map[gEdgePathMarker, e1]]];
  m2 = Union[Flatten[Map[gEdgePathMarker, e2]]];

  res = {};
  If[(m1 ≠ {}) ∧ (m2 ≠ {}),
   v1 = Union[
     Flatten[Map[gEdgeValuation, Select[e1, gEdgePathMarker[#] ≠ {} &]]]];
   v2 = Union[Flatten[Map[gEdgeValuation,
       Select[e2, gEdgePathMarker[#] ≠ {} &]]]];

   (* Check first for m1 ⊇ m2, then for m1 ⊆ m2
    *)
   If[(m1 ⋂ m2) == m2,
    res = res ⋃ {{v1, v2}}
   ];
   If[(m1 ⋂ m2) == m1,
    res = res ⋃ {{v2, v1}}
   ]
  ];

  res
 ]
```

■ vPathRelValuations

This function gets two sets of edges *R1* and *R2* with path markers. The result is a set of valuation relations calculated based on the path markers. If both sets contain identical edges, trivial relations of an edge with itself are omitted and will not be part of the result set. If the edges of at least one set contain no path markers or if no relations based on pathes can be found, the empty set is returned.

```
vPathRelValuations[R1_, R2_] := Module[{i, vv, res},
  If[MemberQ[outputSet, 6],
   Print["⟨6⟩ Resolve path related valuation relations R1 = ",
     MatrixForm[R1], " ↔  R2 = ", MatrixForm[R2]];
  ];

  (* Initial relation with all elements of R1,R2.
   *)
  res = vResolvePathEquality[R1, R2];
  If[MemberQ[outputSet, 7],
   Print["⟨7⟩ Initial Relation for R1 ↔ R2 : ", MatrixForm[res]]
  ];

  (* All elements of R1 with each element of R2 and
     all elements of R2 with each element of R1.
   *)
  If[R1 ≠ R2,
   vv = Flatten[Map[vResolvePathEquality[R1, {#}] &, R2], 1] ⋃
     Flatten[Map[vResolvePathEquality[R2, {#}] &, R1], 1];
   If[MemberQ[outputSet, 7],
    Print["⟨7⟩ Relations for R1 with each of R2 and reverse : ",
     MatrixForm[Union[vv]]]
   ];
   res = res ⋃ vv;
  ];

  For[i = 1, i ≤ Length[R1], i++,
   (* Get all relations with the i'th element. The complement on the
      set "R" avoids trivial relation with the edge R⟦i⟧ itself.
    *)
   vv = Flatten[Map[
      vResolvePathEquality[{R1⟦i⟧}, {#}] &, Complement[R2, {R1⟦i⟧}]], 1];
   If[MemberQ[outputSet, 7],
    Print["⟨7⟩ Relations for ", R1⟦i⟧, " × R2 : ", MatrixForm[Union[vv]]]
   ];

   (* Add the results to the result set and continue.
    *)
   res = res ⋃ vv
  ];

  res
 ]
```



```
vPathDecCondValuations[R_, dcSet_] := Module[{ier, iet, ief, vr},
  If[MemberQ[outputSet, 6],
   Print["⟨6⟩ Cond./Dec.-Relation for ", dset]
  ];

  ier = cRIE[R, dcSet];
  iet = cSOE[R, dcSet, "T"];
  ief = cSOE[R, dcSet, "F"];
  If[MemberQ[outputSet, 6],
   Print["⟨6⟩ Dec./Cond. → In: ", MatrixForm[ier], "  out-true: ",
    MatrixForm[iet], "  out-false: ", MatrixForm[ief]]
  ];

  vr = vPathRelValuations[ier, iet ⋃ ief] ⋃
    vPathRelValuations[ier, iet] ⋃ vPathRelValuations[ier, ief];

  If[MemberQ[outputSet, 6],
   Print["⟨6⟩ Cond./Dec.-Relations: ", MatrixForm[vr]]
  ];

  vr
 ]
```

■ vPathRelGraph

...creates a valuation-relation graph of program *P* created using additional path information. The function always uses an internal created set of edges with added path-markers no matter if the edge set of *P* already contains path markers or not.

```
vPathRelGraph[P_] := Module[{r2, res},
  r2 = vPathTrace[gR[P], gST[P], Map[gDecisionIdSet, gD[P]], π];

  res = Flatten[Map[vPathRelValuations[
        gInEdges[r2, gNodeId[#]], gOutEdges[r2, gNodeId[#]]] &, gB[P]], 1] ⋃
    vPathRelValuations[gInEdges[r2, gNodeId[gST[P]⟦2⟧]],
     gOutEdges[r2, gNodeId[gST[P]⟦1⟧]]] ⋃
    Flatten[Map[vPathDecCondValuations[r2, gNodeId[#]] &, gC[P]], 1] ⋃
    Flatten[Map[vPathDecCondValuations[r2, gDecisionIdSet[#]] &, gD[P]],
     1];

  res
 ]
```

■ Transformation Relation-Graph

■ Functional Relations

For convenience, the functional relation symbol was incorporated as last part of the edge label (avoiding the need to introduce a new component in the edge definition). This one character represents the *RelationLabel* and is accessed with the function *gRLabel*. If an edge carries no functional relation label, the symbol "=" is returned. So edges without functional relation labels are treated like carrying the functional relation "=". Different to condition-/decision-labels there is no distinction between functional relations of conditions and decisions. In case of a decision, only the edges outgoing from the decision hypernode are relevant, the functional relations along the path inside the decision are not relevant. Like condition-/decision-labels the functional relation is meant to be valid for the outgoing side of a node/hypernode only.

Calculating the relation function label for a condition/decision is done by the function *vCondDecRelFunc*. The function automatically determines if the input is a node-index or a index-set representing nodes of a decision. In any cases only the outgoing edges labelled with condition/decision label *l* are taken into account. If there are more than one outgoing edge the function returns a combined relation symbol that is calculated with the rules described above. Remember that for a decision only the outgoing edges are relevant for the function relation but no internal edge is mentioned.

```
vCondDecRelFunc[R_, iv_, l_] := Module[{edgeSet, combSet},
  edgeSet = cSOE[R, iv, l];

  (* Calculate the combination-set of functional relations symbols
     and remove the neutral "=".
   *)
  combSet =
   Complement[Union[Map[gRLabel, Map[gEdgeLabel, edgeSet]]], {"="}];

  (* Calculate the result. Be careful about comparing a set with a
    constant set. The constant set must be converted to a set
    first, otherwise the comparison may fail.
   *)
  If[combSet == {},
   "=",
   If[MemberQ[combSet, "X"] ∨ (combSet ⋂ {"+", "-"} == Union[{"+", "-"}]),
    "X",
    First[combSet]
   ]
  ]
 ]
```

■ Node/Decision Transformation Relation

The function *vNodeTransRelation* is a helper for *vTransRelGraph* to determine the transformation relation of a single node or decision hypernode and its counterpart in the original program (if such a counterpart exists).

```
vNodeTransRelation[uP_, tP_, tRel_, vi1_, vi2_] :=
 Module[{rel, tfr, ffr, ivr1, ivr2, relValSet, oDeg1, oDeg2},
  (*Print["vNodeTransRel for ",vi1," ↔ ",vi2];*)

  rel = tRel;
```



```
  (* First of all get some properties of the two Nodes...
   *)
  tfr = vCondDecRelFunc[gR[tP], vi2, "T"];
  ffr = vCondDecRelFunc[gR[tP], vi2, "F"];
  (*Print["tfr,ffr",MatrixForm[{tfr,ffr}]];*)

  ivr1 = cIVR[gR[uP], vi1];
  ivr2 = cIVR[gR[tP], vi2];
  (*Print["ivr1,ivr2",MatrixForm[{ivr1,ivr2}]];*)

  oDeg1 = cSOD[gR[uP], vi1];
  oDeg2 = cSOD[gR[tP], vi2];
  (*Print["oDeg1,oDeg2: ",MatrixForm[{oDeg1,oDeg2}]];*)

  If[MemberQ[outputSet, 8],
   Print["⟨8⟩ (vNodeTransRelation) P2-node ", vi2, " ⟨IV_R=", ivr2,
    ", Out-deg.=", oDeg2, "⟩ ↔ P1-node ", vi1, " ⟨IV_R=", ivr1,
    ", Out-deg.=", oDeg1, "⟩; functional relation is ", {tfr, ffr}]
  ];

  (* Now check relation from P1 to P2 and add it to "rel", if such a
    relation exists. We only add relations for conditions, because
    further relations for simple nodes should be easy to resolve by
    transitivity.
   *)

  relValSet = Union[Flatten[vRelNodes[rel, ivr2]]];
  (*Print["relValSet: ",relValSet];*)

  If[vSubsetOf[relValSet, ivr1] ,

   (*  Add relation for case IVR(x') ⊇ IVR(x) (x' in P2, x in P1).
    *)
   If[MemberQ[outputSet, 9],
    Print["⟨9⟩    + Outg. relation for ", ivr2, " ⊇ ", ivr1]
   ];

   If[(oDeg1 > 1) ∧ (oDeg2 > 1),
    If[tfr ≠ "X",
     rel =
      rel ⋃ vResolveValRel[cIVS[gR[tP], vi2, "T"], cIVS[gR[uP], vi1, "T"]]
    ];
    If[ffr ≠ "X",
     rel =
      rel ⋃ vResolveValRel[cIVS[gR[tP], vi2, "F"], cIVS[gR[uP], vi1, "F"]]
    ];

    (* Check the type of relation,
     the node of the transformed program has
       compared to the node of the untransformed program.
       If the output is amplified or equal,
     then nothing has to be done. If
      output is reduced, nothing can be stated about the output-
       relations and
        therefore the ⊇-relation must be removed.
        Remark:
         It seems to be very inefficient first to add and then to remove
        a relation instead not to add it. But it is
         not sufficient not to add it,
        because the same relation may have been added
      before. So it needs to be
        actively removed!
```

```
    *)
   If[¬ MemberQ[{"=", "+", "X"}, tfr],
    rel = Complement[rel,
      vResolveValRel[cIVS[gR[tP], vi2, "T"], cIVS[gR[uP], vi1, "T"]]]
   ];
   If[¬ MemberQ[{"=", "+", "X"}, ffr],
    rel = Complement[rel,
      vResolveValRel[cIVS[gR[tP], vi2, "F"], cIVS[gR[uP], vi1, "F"]]]
   ]

   (*, else *)
   (* rel=rel⋃vResolveValRel[
       vOutValuations[gR[tP],vi2],vOutValuations[gR[uP],vi1]]*)

  ] (* If oDeg *)
 ]; (* If vSubsetOf *)

 relValSet = Union[Flatten[vRelNodes[rel, ivr1]]];
 If[vSubsetOf[relValSet, ivr2] ,

  (*  Add relation for case IVR(x) ⩾ IVR(x') (x' in P2, x in P1).
   *)
  If[MemberQ[outputSet, 9],
   Print["⟨9⟩    + Outg. relation for ", ivr1, " ⩾ ", ivr2]
  ];
  If[(oDeg1 > 1) ∧ (oDeg2 > 1),
   If[tfr ≠ "X",
    rel =
     rel ⋃ vResolveValRel[cIVS[gR[uP], vi1, "T"], cIVS[gR[tP], vi2, "T"]]
   ];
   If[ffr ≠ "X",
    rel =
     rel ⋃ vResolveValRel[cIVS[gR[uP], vi1, "F"], cIVS[gR[tP], vi2, "F"]]
   ];

   (* Check the type of relation like in the case IVR(x') ⩾ IVR(x),
      but with the nodes swapped. In this case, the relation must
      be removed in case of amplification.
    *)
   If[¬ MemberQ[{"=", "-", "X"}, tfr],
    rel = Complement[rel,
      vResolveValRel[cIVS[gR[tP], vi1, "T"], cIVS[gR[uP], vi2, "T"]]]
   ];
   If[¬ MemberQ[{"=", "-", "X"}, ffr],
    rel = Complement[rel,
      vResolveValRel[cIVS[gR[tP], vi1, "F"], cIVS[gR[uP], vi2, "F"]]]
   ]

   (* , else *)
   (* rel=rel⋃vResolveValRel[
       vOutValuations[gR[uP],vi1],vOutValuations[gR[tP],vi2]] *)

  ] (* If oDeg *)
 ]; (* If vSubsetOf *)

 rel
]
```

■ Transformation Relation Graph Construction

vTransRelGraph investigates the transformation-relation between the untransformed program represented by aCFG *uP* and the transformed program represented by aCFG *tP*. *tP* must be in the extended transformation form, where each nodes contains a additional *rel-id*-component. The first definition is only a shortcut

```
vTransRelGraph[uP_, tP_, hints_] := vTransRelGraph[uP, tP, hints, False]
```

```
vTransRelGraph[uP_, tP_, hints_, pathBased_] :=
 Module[{wto, valIndex, P1rel, P2rel,
   Trel, explore, expDec, decToProc, vd1, vd2, dfr},

  (* Now start to determine transformation relations by using the
     extended transfed graph for identifying eqivalent statements
     in the transformed and in the non-transformed graph.
   *)
  If[pathBased,
   (* PATH BASED analysis *)
   If[MemberQ[outputSet, 4],
    Print["⟨4⟩ Performing transformation-
       relations analysis using PATH-BASED relation graph"],
   ];
   P1rel = vPathRelGraph[uP];
   P2rel = vPathRelGraph[tP]

   , (* else ... SIMPLE analysis *)
   If[MemberQ[outputSet, 4],
    Print["⟨4⟩ Performing transformation-
       relations analysis using SIMPLE relation graph"]
   ];
   P1rel = vValRelGraph[uP];
   P2rel = vValRelGraph[tP]
  ];

  If[MemberQ[outputSet, 5],
   Print["⟨5⟩ ", Length[P1rel], " local Valuation-Relations in P1: "];
   vPrintRelationSet[P1rel, "      ▫ "];
   Print["⟨5⟩ ", Length[P2rel], " local Valuation-Relations in P2:"];
   vPrintRelationSet[P2rel, "      ▫ "]
  ];

  (* By axiom the output of the start-node and the input of
     the termination is the same for both programs. Also add
     the hints to the initial relations.
   *)
  Trel = P1rel ⋃ P2rel ⋃ hints ⋃
    vResolveValEquality[vOutValuations[gR[uP], gNodeId[gST[uP][[1]]],
     vOutValuations[gR[tP], gNodeId[gST[tP][[1]]]] ⋃
    vResolveValEquality[vInValuations[gR[uP], gNodeId[gST[uP][[2]]],
     vInValuations[gR[tP], gNodeId[gST[tP][[2]]]];
  If[MemberQ[outputSet, 6],
   Print["⟨6⟩ Initial Trans.-Rel. for
      start-/termination-node (equality by definition)..."];
   Print["⟨6⟩ P2-node ", gNodeId[gST[tP][[1]]],
    " ↔ P1-node ", gNodeId[gST[uP][[1]]];
   Print["⟨6⟩ P2-node ", gNodeId[gST[tP][[2]]],
    " ↔ P1-node ", gNodeId[gST[uP][[2]]]
  ];

  (* Now traverse all other nodes in top to bottom order. We use now a
     depth first search order to avoid early evalutation of nodes on a
```

```
    deeper level and to evaluate the longest possible chain before
    evaluating nodes on the same level.
    ??? Not really sure if that is better than bfs order, but I think
        it's better for loops. Hmmm... Why?
 *)
wto = gDfsOrderList[gV[tP], gR[tP], gST[tP]];
explore =
 Flatten[Map[gSelectNodeById[gB[tP]⋃gC[tP], #] &, Rest[wto]], 1];
expDec = gD[tP];
If[MemberQ[outputSet, 6],
 Print["〈6〉 Exploring trans.-relations P1 → P2 using order ", wto]
];

While[explore ≠ {},
 (* Check, if there is any related node for "v2". If not,
   (gNodeRelId < 0) there is nothing to do.
  *)
 If[gNodeRelId[First[explore]] ≥ 0,
  Trel = vNodeTransRelation[uP, tP, Trel,
    gNodeRelId[First[explore]], gNodeId[First[explore]]];

  (* Check if there is a decision, the processed node is part of
     and determine, if there is a functional identical decision in
     the non-transformed program "uP".
   *)
  vd2 = gSelectDecisionById[expDec, gNodeId[First[explore]]];
  If[vd2 ≠ {},

   vd1 = gSelectDecisionById[gD[uP], gDecisionRelSet[First[vd2]]];
   If[vd1 ≠ {},
    (*Print["Decision found: ",vd1," <- vd1 | vd2 -> ",vd2];*)
    Trel = vNodeTransRelation[uP, tP, Trel,
      gDecisionIdSet[First[vd1]], gDecisionIdSet[First[vd2]]];

    expDec = Complement[expDec, vd2]

   ] (* If decToProc ≠ {} *)

  ] (* If vd1 ≠ {} *)

 ]; (* if gNodeRelId[First[explore]]≥0 *)

 (* Remove the already processed node from explore. We took the
    first and therefore "Rest" will do the job.
  *)
 explore = Rest[explore];

]; (* While *)

If[MemberQ[outputSet, 5],
 Print["〈5〉 ", Length[Complement[Trel, P1rel⋃P2rel]],
  " Final P1 → P2 relations:"];
 vPrintRelationSet[Complement[Trel, P1rel⋃P2rel], "      ▫ "]
];

Trel (* ⋃ hints ??? *)
]
```

■ Investigating Relations Between Valuation-Sets

vRelNodes returns a set of all nodes, reachable from a given node *v*. The argument set *R* holds the edge set of the graph. This function is normally used for the valuation relation graph to determine the related valuations of a given valuation set *v* in a valuation relation graph with relations *R*.

```
vRelNodes[R_, v_] := Module[{II = {}, X = {v}, w},
  (*Print["→→→ RelNodes for node ",v];*)
  While[X ≠ {},
   w = First[X];
   (*Print["w: ",w,"    X: ",X];*)
   II = II⋃{w}(*⋃Partition[w,1]*);
   (*Print["II: ",II,"   Succ. of ",w,": ",gOutEdges[R,w]];*)
   X = X⋃Map[gEdgeTail[#] &, gOutEdges[R, w]]⋃Partition[w, 1];
   (*Print["X: ",X,"    X\II: ",Complement[X,II]];*)
   X = Complement[X, II]
  ];
  (*Print["←←← ",II];*)
  II
 ]
```

Checking, wether or not a given valuation set δ_i is a superset of another valuation set δ_j can be done by checking if δ_j is a member in the set *vRelNodes(R, {δ_i})*. This is sufficient for most cases especially when checking coverage preservation. In some rare cases this simple procedure may fail. Particularly during construction of a relation graph the presence of obvious relations may be sensitive to the order of the construction sequence. So when using the relations constructed so far the necessary relation may not be available direclty, but other relations may be present which can substitude the lack of relations in the graph. E.g.: If it is necessary to check, if $\delta_i \supseteq \delta_j \cup \delta_k$, then with the simple method a node for δ_i and $\delta_j \cup \delta_k$ must be present. If the node $\delta_j \cup \delta_k$ is missing, the check fails. But if the relations $\delta_i \supseteq \delta_j$ and $\delta_i \supseteq \delta_k$ are both available instead, the relation above can be verified even though successful.

This task is performed by the function *vSubsetOf*. The function takes as parameters a (flat) set of valuation-candidates *sc* created in relation to the superset-valuation and the valuation *v* which should be checked. The function checks the candidates in *sc*, if there are members which imply, that *v* is a subset of the original superset-valuation. *True* or *False* is returned as result.

```
vSubsetOf[sc_, v_] := Module[{},
  (*Print["SubsetOf: ",sc," ⋂ ",v," ⇒ ",sc⋂v];*)
  sc⋂v == v
 ]
```

The operation *Union[Flatten[…]]* in the valuation notation is the union of all valuation-sets and constructs the set δ' from above while the union representation used for valuation-sets is identical to the notation of ρ'.

Preservation Proofs

■ Statement Coverage

Definition: Preservation of Statement Coverage: $\forall b' \in B_2\ \exists b \in B_1$ with $IV_R(b') \supseteq IV_R(b)$. We implement this rule in an algorithmic way by scanning through each basic block of Program 2 and check if there exists any basic block in Program 1 with a valuation label that fullfills the required $\supseteq$ relation.

SCPres checks for Program *P1* and the transformed Program *P2* if the SC-Preservation criteria is fullfilled. Properties of the transformation are defined by *TR* in form of relations between the valuation relation graphs of P1 and P2.

WARNING-Messages:
"Empty Reachability Valuation for node # !!!" will be printed if IV_R returns the empty set. Possible reason: The start node is member in *B* or a non connected node is included in set *B*.

```
cSCPres[P1_, P2_, RG_] := Module[{S1, S2, R1, R2, i, vr, rvr, rn, scpf = True},
   (* To make sure, not to deal with the empty IVR-set we exclude
      start and termination node from the basic block set.
    *)
   S1 = Map[gNodeId, gB[P1]];
   R1 = gR[P1];
   S2 = Map[gNodeId, gB[P2]];
   R2 = gR[P2];

   If[MemberQ[outputSet, 2],
    Print["• SC-Preservation, checking: ∀b'∈ ",
     S2, " ∃b∈B1 with IVR(b')⊇IVR(b)"]
   ];

   (* Scan through each node of P2...
    *)
   For[i = 1, i ≤ Length[S2], i++,
    (* Determine the Reachability Valuation of the current node and their
      related valuations.
     *)
    vr = cIVR[R2, S2[[i]]];
    If[(vr == {}) ∧ (MemberQ[outputSet, 4]),
     Print[
      "⟨4⟩: WARNING! Empty Reachability Valuation for node ", S2[[i]], "!!!"]
    ];

    rvr = Union[Flatten[vRelNodes[RG, vr]]];
    If[MemberQ[outputSet, 6],
     Print["⟨6⟩: P2-node ",
      S2[[i]], " IVR==", vr, " → related valuations: ", rvr]
    ];

    (* Select Nodes of Program 1
      with related valuations and check if there is any.
     *)
    rn = Select[S1, vSubsetOf[rvr, cIVR[R1, #]] &];
    If[MemberQ[outputSet, 3],
     Print["✓ P1-node(s) b with  IVR(", S2[[i]], ") ⊇ IVR(b): ", rn]
    ];

    scpf = scpf ∧ (rn ≠ {});
   ];

   (* Return the accumulated result. *)
   scpf
  ]
```


▪ touchesID

Definition of *touches_ID*: touches_ID(x,ID):⟺ $(IV_T(x) \subseteq ID) \vee (IV_F(x) \subseteq ID)$

touchesID is implemented a little bit different compared to the original definition. Instead of *ID* the predicate gets the flat set *vr* of valuations, related to *ID*. The program object *x* can be a condition or a decision. The difference between condition and decission is handled by the IVS-function. The function is not able to check, wether the given execution-relation *R* is valid for *x* since *x* can be a isolated node.

```
touchesID[R_, x_, vr_] := Module[{ivt, ivf, re},
  ivt = cIVS[R, x, "T"];
  ivf = cIVS[R, x, "F"];
  re = vSubsetOf[vr, ivt] ∨ vSubsetOf[vr, ivf];
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩: [IVT(", x, ")==",
    ivt, " ∨ IVF(", x, ")==", ivf, "] ⊆ ", vr, " → ", re]
  ];

  re
 ]
```

▪ Condition Coverage

Definition of *Condition Coverage*: ∀ $c' \in C(P_2)$ ($\exists\, c \in C(P_1)$ with touches_ID $(c, IV_T(c'))$ and $\exists\, c \in C(P_1)$ with touches_ID $(c, IV_F(c'))$)

```
cCCPres[P1_, P2_, RG_] :=
 Module[{C1, C2, R1, R2, i, vrt, vrf, rvrt, rvrf, rnt, rnf, scpf = True},
   (* To adapt to future enhancements we read some informations
     from left to right and other from right to left. *)
   C1 = Map[gNodeId, gC[P1]];
   R1 = gR[P1];
   C2 = Map[gNodeId, gC[P2]];
   R2 = gR[P2];

   If[MemberQ[outputSet, 2],
    Print["• CC-Preservation, checking ∀ c₂ ∈ ", C2, ": ∃ c ∈ C₁ |
       touches_ID (c, IV_T(c₂)) <and> ∃ c ∈ C₁ | touches_ID(c,IV_F(c₂))"];
    Print["      touches_ID(x,ID):⟺(IV_T(x)⊆ID)∨(IV_F(x)⊆ID)"]
   ];

   (* Scan through each condition node of P2...*)
   For[i = 1, i ≤ Length[C2], i++,
    (* Determine the Satisfyability
      Valuations of the current node and their
       related valuations.
     *)
    vrt = cIVS[R2, C2⟦i⟧, "T"];
    vrf = cIVS[R2, C2⟦i⟧, "F"];
    If[(vrt == {}) ∨ (vrf == {}) ∧ MemberQ[outputSet, 4],
     Print["⟨4⟩: WARNING! At least one empty
        Satisfyability Valuation for node ", C2⟦i⟧, " detected!!!"]
    ];

    rvrt = Union[Flatten[vRelNodes[RG, vrt]]];
    rvrf = Union[Flatten[vRelNodes[RG, vrf]]];
    If[MemberQ[outputSet, 6],
     Print["⟨6⟩ P2: IV_T(",
      C2⟦i⟧, ")==", vrt, " → related valuations ", rvrt];
     Print["⟨6⟩ P2: IV_F(", C2⟦i⟧, ")==", vrf,
      " → related valuations ", rvrf]
    ];

    (* Select Nodes of Program 1
      with related valuations and check if there is any.
     *)
    (*rn=Select[C1,touchesID[R1,#,rvrt]∧touchesID[R1,#,rvrf]&];*)
    rnt = Select[C1, touchesID[R1, #, rvrt] &];
    rnf = Select[C1, touchesID[R1, #, rvrf] &];
    scpf = scpf ∧ (rnt ≠ {}) ∧ (rnf ≠ {});
    If[MemberQ[outputSet, 3],
     Print["✓ P1-node(s): {c | touches_ID (c, IV_T(", C2⟦i⟧, "))} → ", rnt,
      "  <and>  {c | touches_ID (c, IV_F(", C2⟦i⟧, "))} → ", rnf]
    ]
   ];

   (* Return the accumulated result. *)
   scpf
  ] (* CCPres *)
```


▪ Decision Coverage

Definition of *Decision Coverage*: ∀ d' ∈ $D(P_2)$ (∃ d ∈ $D(P_1)$ with touches_ID $(d, \mathrm{IV}_T(d'))$ and ∃ d ∈ $D(P_1)$ with touches_ID $(d, \mathrm{IV}_F(d'))$
Definition of *touches_ID*: as above but in this case x is a decision.

```
cDCPres[P1_, P2_, RG_] :=
 Module[{D1, D2, R1, R2, i, vrt, vrf, rvrt, rvrf, rdt, rdf, scpf = True},
   (* To adapt to future enhancements we read some informations
     from left to right and other from right to left. *)
   D1 = Map[gDecisionIdSet, gD[P1]];
   R1 = gR[P1];
   D2 = Map[gDecisionIdSet, gD[P2]];
   R2 = gR[P2];

   If[MemberQ[outputSet, 2],
    Print["• DC-Preservation, checking ∀ d₂ ∈ ", D2, ": ∃ d ∈ D₁ |
       touches_ID (d, IV_T(d₂)) <and> ∃ d ∈ D₁ | touches_ID(d,IV_F(d₂))"];
    Print["      touches_ID(x,ID):⟺(IV_T(x)⊆ID)∨(IV_F(x)⊆ID)"]
   ];

   (* Scan through each condition node of P2...*)
   For[i = 1, i ≤ Length[D2], i++,
    (* Determine the Satisfyability
      Valuations of the current decision and their
       related valuations.
     *)
    vrt = cIVS[R2, D2⟦i⟧, "T"];
    vrf = cIVS[R2, D2⟦i⟧, "F"];

    If[(vrt == {}) ∨ (vrf == {}) ∧ MemberQ[outputSet, 4],
     Print["⟨4⟩: WARNING! At least one empty Satisfyability
        Valuation for decision ", D2⟦i⟧, "detected!!!"]
    ];
    rvrt = Union[Flatten[vRelNodes[RG, vrt]]];
    rvrf = Union[Flatten[vRelNodes[RG, vrf]]];
    If[MemberQ[outputSet, 6],
     Print["⟨6⟩ P2: IV_T(", D2⟦i⟧, ")==", vrt, " is related with ", rvrt];
     Print["⟨6⟩     IV_F(", D2⟦i⟧, ")==", vrf, " is related with ", rvrf]
    ];

    (* Select Nodes of Program 1
      with related valuations and check if there is any.
     *)
    (*rd=Select[D1,touchesID[R1,#,rvrt]∧touchesID[R1,#,rvrf]&];*)
    rdt = Select[D1, touchesID[R1, #, rvrt] &];
    rdf = Select[D1, touchesID[R1, #, rvrf] &];
    scpf = scpf ∧ (rdt ≠ {}) ∧ (rdf ≠ {});
    If[MemberQ[outputSet, 3],
     Print["✓ P1-Decision(s): {d |  touches_ID (d, IV_T(", D2⟦i⟧, "))} → ",
      rdt, "  <and>  {d | touches_ID (d, IV_F(", D2⟦i⟧, "))} → ", rdf]
    ]
   ];

   (* Return the accumulated result. *)
   scpf
  ] (* DCPres *)
```

■ Modified Condition Decision Coverage

To solve the MCDC preservation problem we have to show among others, that subsets ID1 and ID2 exist, that fullfill the MCDC condition for some condition/decision pair in P1 and in P2. To get a list of possible candidates for these sets, we use the first two parts of the preservation condition to gain characterizations, that must be fullfilled, if these sets exist.

The construction of possible characterizations for ID-sets is based on paths through a decision d, starting at $IV_R(d)$ and terminating at $IV_T(d)$ or $IV_F(d)$. To find these paths, the g*PathSet* function from above is used together with the helper function c*DecisionPath* to check, wether or not a path p fits a certain decision outcome dr. A path fits a certain decision outcome $\gamma \in \{T,F\}$, if the decision labels on all edges in the path are ξ , X or empty, and if γ and ξ are members of the same set determined by the global variable *DecisionTrueFalseSet*.

```
cDecisionPath[p_, dr_] := Module[{pl},
  pl = Complement[vDecisionLabelSet[p], {"", "X"}];
  symbSet =
   If[dr == "X", DecTrueFalseSet, Select[DecTrueFalseSet, MemberQ[#, dr] &]];

  (Length[pl] == 0) ∨
   ((Length[pl] == 1) ∧ (MemberQ[Flatten[symbSet], First[pl]]))
 ]
```

cVariantCondition is a helper to check the validity of a path through a decision. It checks, if a path p includes the same edge as e but with a different condition result. This function is normally used as a helper for examining two paths for common variant conditions (see *cCommonVariantConditions* below). *cVariantCondition* returns a list with the head identifier of e, if an edge with the same head and tail but with a different condition result exists in p. Otherwise, it returns the empty set.

```
cVariantCondition[e_, p_] :=
  Map[gEdgeHead, Select[p, gEdgeHead[e] == gEdgeHead[#] ∧
      gCLabel[gEdgeLabel[e]] ≠ gCLabel[gEdgeLabel[#]] &]];
```

cValidMCDCPath compares two paths *p1* and *p2* if exactly one of the shared conditions has a different condition result. If such a condition exists and if this condition has the identifier c, the function returns *True*, otherwise it returns *False*.

```
cValidMCDCPath[c_, p1_, p2_] := Module[{vc},
  vc = Apply[Union, Map[cVariantCondition[#, p2] &, p1]];
  Length[vc] == 1 ∧ First[vc] == c
 ]
```

InvExprPath performs the *invariant expression* check for a condition c in relation to a true-path tp and a false-path fp. The check is done by intersecting the true-edges and the false-edges with the given paths and checking if at least one result is empty.

```
cInvExprPath[R_, c_, tp_, fp_] := Module[{te, fe},
  te = cSOE[R, c, "T"];
  fe = cSOE[R, c, "F"];

  ((te ⋂ tp == {}) ∨ (fe ⋂ fp == {})) ∧ ((te ⋂ fp == {}) ∨ (fe ⋂ tp == {}))
 ]
```


The function *constructIDcand* returns a list of possible characterizations for ID-sets related to decision d and condition c in the program defined by execution relation R. The characterization for a ID set related to decision d and condition c is a 3-tuple (γ, δ, σ), where γ is the valuation set related to c (determined with $IV_T(c)$ or $IV_F(c)$ in the *mult_control_expr* condition), δ the valuation set related to d (determined by $IV_T(d)$ or $IV_F(d)$ in *mult_control_expr*) and σ is a set containing all valuation sets in all condition $c'' \neq c$ with an empty intersection $ID \bigcap IV_T(c'')$ or $ID \bigcap IV_F(c'')$, determined by the *isInvariantExpr* condition of *uniqueCause*.

The construction of this list is based on analysis of pathes starting at $IV_R(d)$ and terminating at $IV_T(d)$ or $IV_F(d)$ (d is a decision). The edge sequences for the *true* and *false* paths are constructed in a 2 step process. In the first step the pathes from $IV_R(d)$ up to $IV_R(c)$ and from $IV_R(c)$ to $IV_T(d)$ or $IV_F(d)$ are calculated. In the second step all valid combinations of the path torsos calculated in the first step are constructed. The valuation-sets of the edges are used to determine the characteristics of the required valuation-sets ID1/ID2.

```
constructIDcand[R_, d_, c_] :=
 Module[{ect, ecf, edt, edf, de, ce, dEntry, dpt, dpf,
   it, if, jt, jf, cpt, cpf, invCond, e2tf, invVal, ids = {}},

  If[MemberQ[outputSet, 4],
   Print[
    "〈4〉 construct ID candidates for condition ", c, " in decision ", d]
  ];

  (* Determine the outgoing edges and reachability edges for d and c
   *)
  ect = cSOE[R, c, "T"];
  ecf = cSOE[R, c, "F"];
  edt = cSOE[R, d, "T"];
  edf = cSOE[R, d, "F"] ;

  de = cRIE[R, d];
  ce = cRIE[R, c];

  If[MemberQ[outputSet, 5],
   Print["〈5〉 ▫ Condition involves edges IVR → ", MatrixForm[ce],
    "   〈IVT,IVF〉 → 〈", MatrixForm[ect], ",", MatrixForm[ecf], "〉"];
   Print["〈5〉 ▫ Decision involves edges IVR → ", MatrixForm[de],
    "   〈IVT,IVF〉 → 〈", MatrixForm[edt], ",", MatrixForm[edf], "〉"]
  ];

  (* Construct path-
    segments. From IVR(d) to IVR(c) we remove the entry edges,
   because they may disturb qualification as a valid decision path,if they
    contain a decision label from a other decision. Then construct paths
    from IVT(c) to IVT(d) and IVF(c) to IVF(d)
    and select only valid ones.
   *)
  dEntry = Select[Union[Map[Complement[#, de] &, gPathSet[R, de, ce]]],
    cDecisionPath[#, "X"] &];
  If[MemberQ[outputSet, 5],
   Print["〈5〉 Entry paths from IVR(",
    d, ") to IVR(", c, ") ⇒ ", MatrixForm[dEntry]]
  ];

  (* We start with the combinations Condition True/False -
    Decision True/False!
   *)
  dpt = Select[gPathSet[R, ect, edt], cDecisionPath[#, "T"] &];
  dpf = Select[gPathSet[R, ecf, edf], cDecisionPath[#, "F"] &];
```

```
If[MemberQ[outputSet, 5],
 Print["〈5〉 Searching for valid T/F-T/F outcome combinations ⇒"];
 Print["〈5〉 Dec.-True  Path: ", dpt];
 Print["〈5〉 Dec.-False Path: ", dpf]
];

(* Construct all possible combinations of path-elements using a path
   from "dEntry" as the beginning
   and continuing with a path from "dpt" and "dpf"
   as true and false fork of the condition.
    "i" loops over all entry pathes, jt/jf over all true/false pathes.
 *)
For[it = 1, it ≤ Length[dEntry], it++,
 For[if = 1, if ≤ Length[dEntry], if++,
  For[jt = 1, jt ≤ Length[dpt], jt++,
   For[jf = 1, jf ≤ Length[dpf], jf++,

    (* Combine two pieces to
      form the complete path for True ("cpt") and the
      complete path for False ("cpf").
     *)
    cpt = Join[dEntry⟦it⟧, dpt⟦jt⟧];
    cpf = Join[dEntry⟦if⟧, dpf⟦jf⟧];
    If[MemberQ[outputSet, 6],
     Print[
      "〈6〉 ▫ Complete paths to check: True → ", cpt, "   False → ", cpf];
     Print["      Valid decision path and MCDCPath for ",
      c, " ? ", {cDecisionPath[cpt, "T"],
       cDecisionPath[cpf, "F"], cValidMCDCPath[c, cpt, cpf]}]
    ];
    (* Check first, if the path combinations are a valid decision paths.
     *)
    If[cDecisionPath[cpt, "T"] ∧
      cDecisionPath[cpf, "F"] ∧ cValidMCDCPath[c, cpt, cpf],
     (* The control_expr predicate is fullfilled
       by construction. To determine
       the empty intersection we have to do the invariant_expr check.
      *)
     invCond = Select[d, cInvExprPath[R, #, cpt, cpf] &];
     If[MemberQ[outputSet, 6],
      Print["〈6〉 ▫ Invariant Conditions: ", invCond]
     ];

     (* If not all conditions from the complement d\{c} are invariant,
      then this is not a valid path combination and therefore ignore it.
      *)
     If[Complement[d, {c}] == invCond,
      e2tf = Flatten[Map[cSOE[R, #, "T"] ⋃ cSOE[R, #, "F"] &, invCond], 1];
      invVal = Map[{gEdgeValuation[#]} &,
        Select[e2tf, ¬ MemberQ[cpt, #] ∧ ¬ MemberQ[cpf, #] &]];
      If[MemberQ[outputSet, 6],
       Print["〈6〉 + Edges of inv. Cond.: ",
        e2tf, "   invariant valuations: ", invVal]
      ];

      (* If there are invariant conditions
        but no invariant valuation have been found,
       then the invariant_expr predicate is not
         fullfilled. Otherwise add the two
         pathes to the result-list.
       *)
      If[(invCond == {}) ∨ (invVal ≠ {}),
```



```
       ids = ids ⋃ {{{gEdgeValuation[First[dpt⟦jt⟧]]},
           {gEdgeValuation[Last[dpt⟦jt⟧]]}, invVal}} ⋃ {{{gEdgeValuation[
             First[dpf⟦jf⟧]]}, {gEdgeValuation[Last[dpf⟦jf⟧]]}, invVal}};
       If[MemberQ[outputSet, 6],
        Print["〈6〉 + Updated result-list: ", MatrixForm[ids]]
       ]
      ] (* If invCond ∨ invVal *)
     ] (* If invCond *)
    ] (* If valid decision path *)
   ] (* For jf *)
  ] (* For jt *)
 ] (* For if *)
]; (* For it *)

(* Now trying the combinations Condition True/False -
  Decision False/True!
 *)
dpt = Select[gPathSet[R, ecf, edt], cDecisionPath[#, "T"] &];
dpf = Select[gPathSet[R, ect, edf], cDecisionPath[#, "F"] &];
If[MemberQ[outputSet, 5],
 Print["〈5〉 Searching for valid T/F-F/T outcome combinations ⇒"];
 Print["〈5〉 Dec.-True  Path: ", dpt];
 Print["〈5〉 Dec.-False Path: ", dpf]
];

(* Construct again all possible combinations of path-
  elements using a path
   from "dEntry" as the beginning and continuing
   with a path from "dpt" and "dpf"
   as true and false fork of the condition.
    "i" loops over all entry pathes, jt/jf over all true/false pathes.
 *)
For[it = 1, it ≤ Length[dEntry], it++,
 For[if = 1, if ≤ Length[dEntry], if++,
  For[jt = 1, jt ≤ Length[dpt], jt++,
   For[jf = 1, jf ≤ Length[dpf], jf++,

    (* Combine two pieces to
      form the complete path for True ("cpt") and the
      complete path for False ("cpf").
     *)
    cpt = dEntry⟦it⟧ ⋃ dpt⟦jt⟧;
    cpf = dEntry⟦if⟧ ⋃ dpf⟦jf⟧;
    If[MemberQ[outputSet, 6],
     Print[
      "〈6〉 ▫ Complete paths to check: True → ", cpt, "   False → ", cpf];
     Print["      Valid decision path and MCDCPath for ",
      c, " ? ", {cDecisionPath[cpt, "T"],
       cDecisionPath[cpf, "F"], cValidMCDCPath[c, cpt, cpf]}]
    ];

    (* Check first, if the path combination is a valid decision path.
     *)
    If[cDecisionPath[cpt, "T"] ∧
      cDecisionPath[cpf, "F"] ∧ cValidMCDCPath[c, cpt, cpf],
     (* The control_expr predicate is fullfilled
       by construction. To determine
       the empty intersection we have to do the invariant_expr check.
      *)
     invCond = Select[d, cInvExprPath[R, #, cpt, cpf] &];
     If[MemberQ[outputSet, 6],
      Print["〈6〉 ▫ Invariant Conditions: ", invCond]
```

```
      ];

      (* If not all conditions from the complement d\{c} are invariant,
       then this is not a valid path combination and therefore ignore it.
       *)
      If[Complement[d, {c}] == invCond,
       e2tf = Flatten[Map[cSOE[R, #, "T"] ⋃ cSOE[R, #, "F"] &, invCond], 1];
       invVal = Map[{gEdgeValuation[#]} &,
         Select[e2tf, ¬ MemberQ[cpt, #] ∧ ¬ MemberQ[cpf, #] &]];
       If[MemberQ[outputSet, 6],
        Print["〈6〉 + Edges of inv. Cond.: ",
         e2tf, "   invariant valuations: ", invVal]
       ];

       (* If there are invariant conditions
         but no invariant valuation have been found,
        then the invariant_expr predicate is not
          fullfilled. Otherwise add the two
          pathes to the result-list.
        *)
       If[(invCond == {}) ∨ (invVal ≠ {}),
        ids = ids ⋃ {{{gEdgeValuation[First[dpt⟦jt⟧]]},
            {gEdgeValuation[Last[dpt⟦jt⟧]]}, invVal}} ⋃ {{{gEdgeValuation[
              First[dpf⟦jf⟧]]}, {gEdgeValuation[Last[dpf⟦jf⟧]]}, invVal}};
        If[MemberQ[outputSet, 6],
         Print["〈6〉 + Updated result-list: ", MatrixForm[ids]]
        ]
       ] (* If invCond ∨ invVal *)
      ] (* If invCond *)
     ] (* If valid decision path *)
    ] (* For jf *)
   ] (* For jt *)
  ] (* For if *)
 ]; (* For it *)

 ids
]
```

Additional Predicates used in Preservation-Condition for *Modified Condition Decision Coverage* (MCDC):

mult_control_expr (ID_1, ID_2, x) : $\Longleftrightarrow$ ($ID_1 \subseteq IV_T(x) \ \wedge \ ID_2 \subseteq IV_F(x)$) $\vee$ ($ID_1 \subseteq IV_F(x) \ \wedge \ ID_2 \subseteq IV_T(x)$)

The implementation *multControlExpr* takes the execution relation *R* of the program and the valuation relation graph *rg* as additional parameter. Since this predicate is always used in a way like "$\exists\ ID_1, ID_2 \subseteq \mathbb{ID}$ such that *mult_control_expr* is true" we take the immediate supersets of ID1 and ID2 as a substitude for ID1 and ID2 as parameters and check, if IV_T and IV_F are in a $\supseteq$-relation. Because *IVS* can handle condition-input and decision-input for *x*, this version of *multControlExpr* can handle conditions as well as decisions.


```
cMultControlExpr[R_, rg_, ID1_, ID2_, x_] := Module[{vst, vsf, ivt, ivf, ok},
  If[MemberQ[outputSet, 4],
   Print["〈4〉 MultControlExpr: {ID₁}: ",
    ID1, "   {ID₂}: ", ID2, "   Cond./Dec. checked: ", x]
  ];

  vst = cIVS[R, x, "T"];
  vsf = cIVS[R, x, "F"];
  ivt = vRelNodes[rg, vst];
  ivf = vRelNodes[rg, vsf];

  If[MemberQ[outputSet, 3],
   Print["         ∘ IVₜ(", x, ")==", vst, " ↔ ", ivt];
   Print["         ∘ IVF(", x, ")==", vsf, " ↔ ", ivf]
  ];

  (* Check, if valuations of ID1 and ID2 satisfy the membership-
    relation with ivt and ivf
   *)
  ok = (MemberQ[ivt, ID1] ∧ MemberQ[ivf, ID2]) ∨
    (MemberQ[ivf, ID1] ∧ MemberQ[ivt, ID2]);

  If[MemberQ[outputSet, 5],
   Print["〈5〉 + ID₁ ⊆ IVₜ(", x, ") ∧ ID₂ ⊆ IVF(",
    x, ") ∨ ID₁ ⊆ IVF(", x, ") ∧ ID₂ ⊆ IVₜ(", x, ")"];
   Print["〈5〉 + (", MemberQ[ivt, ID1], " ∧ ", MemberQ[ivf, ID2],
    ") ∨ (", MemberQ[ivf, ID1], " ∧ ", MemberQ[ivt, ID2], ") ⇒ ", ok]
  ];

  ok
 ]
```

In the original definition of the predicate *isInvariantExpression*, ID is a set {*id1*,*id2*} of pairs of elements, with *id1* ∈ ID1 and *id2* ∈ ID2. It is normally used in a way that all possible combinations of elements of the original sets ID1 and ID2 have to satisfy this condition. In this implementation, each argument *ID1* and *ID2* holds a list of valuations. Instead of checking for intersections with {*id1*,*id2*} we check the whole base sets for non-strict super-set. If this is *true*, then the required element combinations {*id1*,*id2*} considered above must exist.

```
cIsInvariantExpr[R_, rg_, ID1_, ID2_, x_] :=
 Module[{xID1, xID2, ivt, ivf, i, rn, r1, r2, r3, r4, re = True},

  (* Add the union of all subsets listed in ID1 and ID2, because
    the empty subset condition must also be true for the union
    if it is true for all of the subsets.
   *)
  xID1 = ID1 ⋃ {Union[Flatten[ID1]]};
  xID2 = ID2 ⋃ {Union[Flatten[ID2]]};

  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ isInvariantExpr for Node: ",
    x, " in rel. with xID1: ", xID1, "   xID2: ", xID2]
  ];

  ivt = cIVS[R, x, "T"];
  ivf = cIVS[R, x, "F"];
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ ▫ IV_T(", x, ") == ", ivt, "   IV_F(", x, ") == ", ivf]
  ];

  (* Check, if there exists at least one valuation set from xID1 and xID2,
     which is a superset of IV_T
   *)
  r1 = Select[xID1, MemberQ[vRelNodes[rg, #], ivt] &];
  r2 = Select[xID2, MemberQ[vRelNodes[rg, #], ivt] &];

  (* Check, if there exists at least one valuation set from ID1 and ID2,
     which is a superset of IV_F
   *)
  r3 = Select[xID1, MemberQ[vRelNodes[rg, #], ivf] &];
  r4 = Select[xID2, MemberQ[vRelNodes[rg, #], ivf] &];

  (* Now check, if at least two corresponding sets are not empty
   *)
  re = ((r1 ≠ {}) ∧ (r2 ≠ {})) ∨ ((r3 ≠ {}) ∧ (r4 ≠ {}));

  If[MemberQ[outputSet, 5],
   Print["⟨5⟩ + (", r1, "≠{} ∧ ",
    r2, "≠{}) ∨ (", r3, "≠{} ∧ ", r4, "≠{}) ⇒ ", re]
  ];

  re
 ] (* Module *)
```

control_expr(td_1, td_2, x) :⟺ ($td_1 \in IV_T(x) \land td_2 \in IV_F(x)$) ∨ ($td_2 \in IV_T(x) \land td_1 \in IV_F(x)$)

See remark on *unique_cause*: We do not implement this predicate but using *mult_control_expr* instead.

unique_cause(c_1, d, td_1, td_2) :⟺
control_expr (td_1, td_2, c_1) ∧ control_expr (td_1, td_2, d) ∧ ∀ $c_2 \in C(d)$ $(c_2 \neq c_1) \Rightarrow$ is_invariant_expr ({ td_1, td_2 }, c_2)

Since *unique_cause* is always used in a way like "∀ ⟨ id_1, id_2⟩ ∈ $ID_1 \times ID_2$ unique_cause (c, d, id_1, id_2)" we define this predicate and all its sub-predicates slightly different. Instead of using all possible combinations of pairs of elements and checking $td_x \in Y$ we take the whole set TD_x and check $TD_x \subseteq Y$. If this succeeds, the original proposition of *unique_cause* must be true because of the definition of ⊆ (A⊆B :⟺ ∀ a∈A ⟹ a∈B).

To get higher efficiency, we also do not use single sets for TD_1 and TD_2 but take the whole candidate list and make an intersection with the related set list of IV_T and IV_F. Normally both candidate lists will be the same, but for any reason whatever we distinguish between the two candidate lists.

Since the sets, fulfilling the condions must be always the same, we use the following procedure: Each predicate returns a list with the candidates that fulfill the predicates conditions. This reduced list is then used as input for the next predicate. The test was successful, if after checking the last predicate at least one candidate is left.

```
cUniqueCause[R_, rg_, d_, c_, TC1_, TC2_] := Module[{rr1, rr2, rr3, dd},
  (* Determine the conditions relevant for the non-variant condition
     check and store them in "dd".
   *)
  dd = Complement[d, {c}];

  (* Calculate the overall result and store it for debugging purposes
   *)
  rr1 = cMultControlExpr[R, rg, TC1[[1]], TC2[[1]], c];
  rr2 = cMultControlExpr[R, rg, TC1[[2]], TC2[[2]], d];
  rr3 = (Select[dd, cIsInvariantExpr[R, rg, TC1[[3]], TC2[[3]], #] &] == dd);

  If[MemberQ[outputSet, 4],
   Print["⟨4⟩ UniqueCause:  d'= ", d,
    ",  c'= ", c, ",   {TC1}: ", TC1, ",   {TC2}: ", TC2,
    " ⇒ (cMCE(c) ∧ cMCE(d) ∧ cIE ∀ :: ", rr1, "^", rr2, "^", rr3]
  ];

  rr1 ∧ rr2 ∧ rr3
 ] (* Module *)
```

(Realistic) Preservation Criteria of *Modified Condition Decision Coverage* (MCDC):

∀ d'∈ D_2 ∀ c'∈ C(d') ∃ $ID_1, ID_2 \subseteq \mathbb{ID}$
(∃d∈ D_1 ∃ c∈ C(d) ∃ $ID_{tmp} \subseteq \mathbb{ID}$ with mult_control_expr (ID_1, ID_{tmp}, c) ∧ ∀ ⟨ id_1, id_2⟩ ∈ $ID_1 \times ID_{tmp}$ unique_cause (c, d, id_1, id_2)
∧
(∃d∈ D_1 ∃ c∈ C(d) ∃ $ID_{tmp} \subseteq \mathbb{ID}$ with mult_control_expr (ID_2, ID_{tmp}, c) ∧ ∀ ⟨ id_1, id_2⟩ ∈ $ID_2 \times ID_{tmp}$ unique_cause (c, d, id_1, id_2)
∧
∀ ⟨ id_1, id_2⟩ ∈ $ID_1 \times ID_2$ unique_cause (c', d', id_1, id_2)

cMCDCPres implements the realistic preservation criterion for MCDC.

```
cMCDCPres[P1_, P2_, RG_] := Module[{D1, D2, R1, R2, i, j, cc, dd,
   c, d, ivrdd, IDcand, rc, found, ci, cj, lengthID, scpf = True},
  D1 = Map[gDecisionIdSet, gD[P1]];
  R1 = gR[P1];
  D2 = Map[gDecisionIdSet, gD[P2]];
  R2 = gR[P2];

  If[MemberQ[outputSet, 2],
   Print[
    "● MCDC-Preservation, checking ∀d'∈D2 ∀c'∈C(d') ∃ ID1,ID2 ⊆ 𝕀𝔻"];
   Print["      (∃ d∈D1 ∃c∈C(d) ∃ IDtmp ⊆ 𝕀𝔻 with mult_control_expr(ID1,
      IDtmp,c) ∧ ∀⟨id1,id2⟩∈ID1×IDtmp unique_cause(c,d,id1,id2) ∧"];
   Print["       ∧ (∃ d∈D1 ∃c∈C(d) ∃ IDtmp ⊆ 𝕀𝔻 with mult_control_expr(ID2,
      IDtmp,c) ∧ ∀⟨id1,id2⟩∈ID2×IDtmp unique_cause(c,d,id1,id2) ∧"];
   Print["       ∧ ∀⟨id1,id2⟩∈ID1×ID2 : unique_cause(c',d',id1,id2)"];
```

```
 Print["        ▫ mult_control_expr(ID₁,ID₂,x):⟺
    (ID₁⊆IV_T(x)∧ID₂⊆IV_F(x))∨(ID₁⊆IV_F(x)∧ID₂⊆IV_T(x))"];
 Print["        ▫ unique_cause(c,d,td₁,td₂):⟺ control_expr
    (td₁,td₂,c) ∧ control_expr(td₁,td₂,d) ∧"];
 Print["               ∧ ∀c₂∈C(d) (c₂≠c₁) ⟹
    is_invariant_expr({td₁,td₂},c₂)"];
 Print["        ▫ control_expr(td₁,td₂,x):⟺ (td₁∈IV_T(
    x)∧ td₂∈IV_F(x)) ∨ (td₂∈IV_T(x) ∧ td₁∈IV_F(x))"];
 Print["        ▫ is_invariant_expr(ID,x):⟺ (ID⋂
    IV_T(x)={}) ∨ (ID⋂IV_F(x)={})"]
];

(* First of all, construct candidates-list for ID1 and ID2 from P1
 *)
If[MemberQ[outputSet, 3],
 Print["✓ Searching for candidates of temporary sets ID₁ and ID₂"]
];

IDcand = {};
For[i = 1, i ≤ Length[D1], i++,
 d = D1[[i]];
 For[j = 1, j ≤ Length[d], j++,
  IDcand = IDcand⋃constructIDcand[R1, d, d[[j]]]
 ] (* For j *)
]; (* For i *)

If[MemberQ[outputSet, 3],
 Print["✓ Final ID₁/ID₂ candidates: ", MatrixForm[IDcand]]
];

(* Scan through each Decision d' of P2 and each Condition c' of d'
   and try to find a pair from candidate-list, that fullfills the
  unique cause condition.
 *)
For[i = 1, i ≤ Length[D2], i++,
 d = D2[[i]];
 If[MemberQ[outputSet, 4],
  Print["⟨4⟩ (", i, "): P2-Decision ", d]
 ];

 For[j = 1, j ≤ Length[d], j++,
  c = d[[j]];
  If[MemberQ[outputSet, 4],
   Print["⟨4⟩  ○(", j, "): P2-Condition ", c]
  ];

  (* Now search for a pair of candidates, that fullfills UniqueCause
   *)
  lengthID = Length[IDcand];
  ci = 1;
  cj = 1;
  found = False;
  While[¬ found ∧ (ci ≤ lengthID),
   found = cUniqueCause[R2, RG, d, c, IDcand[[ci]], IDcand[[cj]]];
   ++cj;
   If[cj > lengthID,
    cj = 1;
    ++ci
   ]
  ];

  scpf = scpf ∧ found;
```



```
   If[¬ found ∧ MemberQ[outputSet, 3],
    Print["*** No ID1/ID2 pair satisfies UniqueCause()!"]
   ];

  ] (* For j *)
 ]; (* For i *)

 scpf

] (* Module *)
```

■ Path Coverage

Since *Scoped Path Coverage* is a quite complex task due to the huge set of possible segmentations of a program we first concentrate on *Path Coverage* inside the program snippet.

■ cUnifiedEdgeSet

This function re-organizes a given edge-set E by removing multiple edges between nodes and joining the labels of the multiple edges into the unified edge (see also *gLabelJoin* for details). Valuation-information and path-information is omitted.

```
cUnifiedEdgeSet[E_] := Module[{explore = E, uE = {}, current},
  While[explore ≠ {},
   current = Select[explore, (gEdgeHead[First[explore]] == gEdgeHead[#]) ∧
       (gEdgeTail[First[explore]] == gEdgeTail[#]) &];

   uE = Append[uE,
     If[Length[current] == 1,
      Take[First[current], {1, 3}],
      {gEdgeHead[First[current]], gEdgeTail[First[current]],
       gLabelJoin[Union[Flatten[Map[gEdgeLabel, current]]]]}
     ]
    ];

   explore = Complement[explore, current]
  ];

  uE
 ]
```

■ cCondPathTrace

This function searches a (unified) path PP and performs a distribution of the found conditions to the sets C_T and C_F, where C_T contains all conditions that contribute to the path with the *True* branch and C_F contains all conditions where the *False* branch is coincident with the path. The result is returned as a tuple $\langle C_T, C_F \rangle$ of the resulting sets.

```
cCondPathTrace[PP_] := {Map[gEdgeHead,
   Select[PP, MemberQ[{"T", "V"}, gCLabel[gEdgeLabel[#]]] &]],
  Map[gEdgeHead, Select[PP, MemberQ[{"F", "V"}, gCLabel[gEdgeLabel[#]]] &]]}
```

■ isCondTFenclosed

Preservation proof for scoped path coverage uses a predicate *is_CondTF_enclosed* with the following definition:

is_CondTF_enclosed(ID, C_T, C_F) :$\Longleftrightarrow$ $\exists\, c_T \in C_T$ with $\mathrm{IV}_T(c_T) \subseteq$ ID **or** $\exists\, c_F \in C_F$ with $\mathrm{IV}_F(c_F) \subseteq$ ID

The implementation *isCondTFenclosed* uses the (flat) set *idr* of related valuation sets instead of *ID*. *ct* and *cf* are sets of condition id's. *R* is the execution relation of the Cfg.

```
isCondTFenclosed[R_, idr_, ct_, cf_] := Module[{ts, fs},
  If[MemberQ[outputSet, 4],
   Print["〈4〉 ▫ isCondTFenclosed: cT∈",
    ct, "   cF∈", cf, " check against ID ⊇ ", idr]
  ];

  ts = Select[ct, vSubsetOf[idr, cIVS[R, #, "T"]] &];
  fs = Select[cf, vSubsetOf[idr, cIVS[R, #, "F"]] &];

  If[MemberQ[outputSet, 5],
   Print["〈5〉 cT ∈ CT | IVT(cT) ⊆ ID: ", ts, " ⇒ ", ts ≠ {}];
   Print["〈5〉 cF ∈ CF | IVF(cF) ⊆ ID: ", fs, " ⇒ ", fs ≠ {}];
  ];

  (ts ≠ {}) ∨ (fs ≠ {})
 ]
```

■ allConditionsEnclosed

This function is a helper that checks *isCondTFenclosed* for a complete set of conditions. It returns *True* if all conditions in *c* fullfill *isCondTFenclosed* for the given path *path* and *False*, if at least one check fails.

```
allConditionsEnclosed[R1_, R2_, vr_, c2_, tf_, path_] :=
 Module[{ct, cf, rv},
  If[MemberQ[outputSet, 6],
   Print["〈6〉 ▫ Checking \"", tf, "\"-conditions ", c2, " on path ", path]
  ];

  {ct, cf} = cCondPathTrace[path];
  res = Map[isCondTFenclosed[R1,
      Union[Flatten[vRelNodes[vr, cIVS[R2, #, tf]]]], ct, cf] &, c2];

  If[MemberQ[outputSet, 5],
   Print["〈5〉 + P1-Condition split CT: ", ct, "    CF: ", cf];
   Print["〈5〉 + isCondTFenclosed ∀c∈",
    c, ": ", res, " ⇒ ", Apply[And, res]]
  ];

  Apply[And, res]
 ]
```

■ cPCPres

This is the main function for Path-Coverage-Preservation check.


```
cPCPres[P1_, P2_, RG_] := Module[
   {scpf = True, C1, R1, uR1, C2, R2, uR2, PP1, PP2, i, ct2, cf2, ppt, ppf},
   (* First of all, get the information most needed from "P1"
      and "P2".
    *)
   C1 = Map[gNodeId, gC[P1]];
   R1 = gR[P1];
   C2 = Map[gNodeId, gC[P2]];
   R2 = gR[P2];

   If[MemberQ[outputSet, 2],
    Print["• PC-Preservation, checking ∀pp'∈PP(P2) ∃pp∈PP(P1) :"];
    Print["      IVR(BS(pp') ⊇ IVR(BS(pp) ∧"];
    Print["       ∧ ∀c'∈CT(pp') ∃pp∈PP(
       P1) : is_condTF_enclosed(IVT(c'),CT(pp),CF(pp)) ∧"];
    Print["       ∧ ∀c'∈CF(pp') ∃pp∈PP(P1) : is_condTF
       _enclosed(IVF(c'),CT(pp),CF(pp))"];
    Print["           is_condTF_enclosed(ID,CT,CF):⟺ ∃cT∈
       CT : IVT(CT) ⊆ ID ∨ ∃cF∈CF : IVF(cF) ⊆ ID"];
   ];

   (* Determine all (unified) pathes in P1 and P2.
    *)
   uR1 = cUnifiedEdgeSet[gR[P1]];
   uR2 = cUnifiedEdgeSet[gR[P2]];
   PP1 = gPathSet[uR1, gOutEdges[uR1, gNodeId[gST[P1][[1]]],
     gInEdges[uR1, gNodeId[gST[P1][[2]]]]];
   PP2 = gPathSet[uR2, gOutEdges[uR2, gNodeId[gST[P2][[1]]],
     gInEdges[uR2, gNodeId[gST[P2][[2]]]]];

   If[MemberQ[outputSet, 3],
    Print["✓ Found ", Length[PP2],
     " pathes in P2 and ", Length[PP1], " pathes in P1."]
   ];

   (* Iterate over all pathes of program "P2". It would be more efficient
      to stop immediately if
     the the accumulated result changes to "False",
      but for documentary reasons we iterate over ALL pathes of "P2".
    *)
   For[i = 1, (i ≤ Length[PP2]) (*∧scpf*), i++,
    If[MemberQ[outputSet, 3],
     Print["✓ (", i, ") P2-Path ", PP2[[i]]
    ];
    (* Determine the condition distribution along
       the selected path.
     *)
    {ct2, cf2} = cCondPathTrace[PP2[[i]]];

    (* Check all conditions of CT of P2 for "isCondTFenclosed"
     *)
    ppt = Select[PP1, allConditionsEnclosed[R1, R2, RG, ct2, "T", #] &];

    If[MemberQ[outputSet, 3],
     Print["✓ Pathes in P1 fullfilling
        isCondTFenclosed(IVT(c2),...) for c2 ∈ ", ct2, ":"];
     Print["   ", MatrixForm[ppt]]
    ];

    ppf = Select[PP1, allConditionsEnclosed[R1, R2, RG, cf2, "F", #] &];
```

```
  If[MemberQ[outputSet, 3],
   Print["✓ Pathes in P1 fullfilling
      isCondTFenclosed(IV_F(c_2),...) for c_2 ∈ ", cf2, ":"];
   Print["   ", MatrixForm[ppf]]
  ];

  (* Add the result for the current path to the overall
    result.
   *)
  scpf = scpf ∧ (ppt ≠ {}) ∧ (ppf ≠ {});
  If[MemberQ[outputSet, 3],
   Print["✓ (", i, ") Preserving PC ⇒ ",
    (ppt ≠ {}) ∧ (ppf ≠ {}), "   (", scpf, ")"]
  ];

  scpf
 ]; (* For i *)

 scpf
]
```

Case Studies

The following use cases present some examples. The assignements define the structures of the aCFG sets B,D,-R,ST and the aCFG structure P itself. Each component is defined twice, one for the non-transformed program and one for the transformed program. If necessary, some supplementary input-valuation set relations are defined.

Naming of Use Cases and naming of aCFG components: A combination of letters and numbers identifies the different components of the aCFG definitions. The first part of each name is an abbreviation of the use-case name. The abbreviation of the name is defined in the headline of the use case. The use-case name is followed by the lower-case letter *u* if the component is related to the non-transformed program. The identifier is followed by the lower-case letter *t* if it refers to the transformed program. One or two upper-case letters at the end of the components name identify the aCFG components: B for the statement set *B*, D for the decision set *D*, R for the edge set *R* and ST for the entry-node/termination-node tuple *ST*. Finally P identifies the structure of the whole aCFG.

Examples: **UceuB** identifies the statement set B of the aCFG of the non-transformed program for the use case "Useless Code Elimination". **LttR** identifies the edge set R of the aCFG of the transformed program for the Loop Tiling use case.

■ Useless Code Elimination (Uce)

Delete code, wich has no influence on the programs final result (e.g. assignment to a variable, which is not used until its next definition).

Original Program:

```
UceuB := {{"stm1", 2}, {"stm2", 3}, {"stm3", 4}}

UceuD := {}

UceuR := {{1, 2, "", δ1}, {2, 3, "", δ2}, {3, 4, "", δ3}, {4, 5, "", δ4}}

UceuST := {{"start", 1}, {"stop", 5}}
```



```
UceuP := eCFG[UceuB, UceuD, UceuR, UceuST]
```

Transformed Program:

```
UcetB := {{"stm1", 12, 2}, {"stm3", 14, 4}}

UcetD := {}

UcetR := {{11, 12, "", ρ1}, {12, 14, "", ρ2}, {14, 15, "", ρ3}}

UcetST := {{"start", 11, 1}, {"stop", 15, 5}}

UcetP := eCFG[UcetB, UcetD, UcetR, UcetST]
```

■ Condition Reordering If-Then-Else with Full Evaluation (Two Conditions with AND) (Tcfr)

Reordering of condition evaluation in a AND-connected two condition two-way branch, where each condition is always executed independent of the outcome of the first condition.

Original Program:

```
TcfruB := {{"then", 3}, {"else", 4}}

TcfruD := {{"decision1", {"cond1", 1}, {"cond2", 2}}}

TcfruS := {"s", 0}

TcfruT := {"t", 5}

TcfruR := {{0, 1, "", δ1}, {1, 2, "TX", δ2}, {1, 2, "FF", δ3}, {2, 3, "TT", δ4},
  {2, 4, "TF", δ5}, {2, 4, "FF", δ6}, {3, 5, "", δ7}, {4, 5, "", δ8}}

TcfruP := {TcfruB, TcfruD, TcfruR, TcfruS, TcfruT}

TcfruP

{{{then, 3}, {else, 4}}, {{decision1, {cond1, 1}, {cond2, 2}}},
 {{0, 1, , δ1}, {1, 2, TX, δ2}, {1, 2, FF, δ3}, {2, 3, TT, δ4},
  {2, 4, TF, δ5}, {2, 4, FF, δ6}, {3, 5, , δ7}, {4, 5, , δ8}}, {s, 0}, {t, 5}}
```

Transformed Program:

```
TcfrtB := {{"then", 13, 3}, {"else", 14, 4}}

TcfrtD :=  {{"decision1", {"cond1*", 11, 2}, {"cond2*", 12, 1}}}

TcfrtS := {"s", 10}

TcfrtT := {"t", 15}

TcfrtR := {{10, 11, "", ρ1}, {11, 12, "TX", ρ2},
  {11, 12, "FF", ρ3}, {12, 13, "TT", ρ4}, {12, 14, "TF", ρ5},
  {12, 14, "FF", ρ6}, {13, 15, "", ρ7}, {14, 15, "", ρ8}}

TcfrtP := {TcfrtB, TcfrtD, TcfrtR, TcfrtS, TcfrtT}
```

The following Relation should be part of the calculated transformation relation:

$\rho_1 == \delta_1$ T1
$\rho_2 == \delta_4 \bigcup \delta_5$ T2
$\rho_3 == \delta_6$ T3
$\rho_6 == \delta_3$ T4
$\rho_4 \bigcup \rho_5 == \delta_2$ T5
$(\rho_7 == \delta_7)$ T6
$(\rho_8 == \delta_8)$ T7

■ Condition Reordering If-Then-Else with Shortcut Evaluation (Two Conditions with AND) (Tcsr)

Reordering of condition evaluation in a AND-connected two condition two-way branch.

Original Program:

```
TcsruB := {{"then", 3}, {"else", 4}}

TcsruD := {{"decision1", {"cond1", 1}, {"cond2", 2}}}

TcsruS := {"s", 0}

TcsruT := {"t", 5}

TcsruR := {{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}}

TcsruP := {TcsruB, TcsruD, TcsruR, TcsruS, TcsruT}
```

Transformed Program:

```
TcsrtB := {{"then", 13, 3}, {"else", 14, 4}}

TcsrtD :=  {{"decision1", {"cond1*", 11, 2}, {"cond2*", 12, 1}}}

TcsrtS := {"s", 10, 0}

TcsrtT := {"t", 15, 5}

TcsrtR := {{10, 11, "", ρ1}, {11, 12, "TX", ρ2}, {11, 14, "FF", ρ3},
  {12, 13, "TT", ρ4}, {12, 14, "FF", ρ5}, {13, 15, "", ρ6}, {14, 15, "", ρ7}}

TcsrtP := {TcsrtB, TcsrtD, TcsrtR, TcsrtS, TcsrtT}
```

The following relations should be included in the result of *vTransRelGraph* :

$(\rho_1 == \delta_1)$ T1
$(\rho_2 \supseteq \delta_4)$ T2
$(\rho_3 \supseteq \delta_5)$ T3
$(\rho_4 \subseteq \delta_2)$ T4
$(\rho_5 \subseteq \delta_3)$ T5
$(\rho_6 == \delta_6)$ T6
$(\rho_7 == \delta_7)$ T7

■ Condition Reordering If-Then with Shortcut Evaluation (Swap two Conditions connected with AND) (Itar)

Reordering of condition evaluation in a AND-connected two condition branch with empty else branch.

Original Program:

```
ItaruB := {{"then", 4}}

ItaruD := {{"IF", {"cond1", 2}, {"cond2", 3}}}

ItaruST := {{"start", 1}, {"stop", 5}}

ItaruR := {{1, 2, "", δ1}, {2, 3, "TX", δ2},
  {2, 5, "FF", δ3}, {3, 4, "TT", δ4}, {3, 5, "FF", δ5}, {4, 5, "", δ6}}

ItaruP := eCFG[ItaruB, ItaruD, ItaruR, ItaruST]
```

Transformed Program:

```
ItartB := {{"then", 14, 4}}

ItartD :=  {{"IF", {"cond2", 12, 3}, {"cond1", 13, 2}}}

ItartST := {{"start*", 11}, {"stop*", 15}}

ItartR := {{11, 12, "", ρ1}, {12, 13, "TX", ρ2}, {12, 15, "FF", ρ3},
  {13, 14, "TT", ρ4}, {13, 15, "FF", ρ5}, {14, 15, "", ρ6}}

ItartP := eCFG[ItartB, ItartD, ItartR, ItartST]
```

The following relations should be included in the result of *vTransRelGraph*:

$(\rho_1 == \delta_1)$ T1
$(\rho_2 \supseteq \delta_4)$ T2
$(\rho_3 \supseteq \delta_5)$ T3
$(\rho_4 \subseteq \delta_2)$ T4
$(\rho_5 \subseteq \delta_3)$ T5
$(\rho_6 == \delta_6)$ T6
$(\rho_5 == \delta_5)$ T7

■ Condition Reordering If-Then with Shortcut Evaluation (Swap two Conditions connected with OR) (Itor)

Reordering of condition evaluation in a OR-connected two condition branch with empty else branch.

Original Program:

```
ItoruB := {{"then", 4}}

ItoruD := {{"IF", {"cond1", 2}, {"cond2", 3}}}

ItoruST := {{"start", 1}, {"stop", 5}}

ItoruR := {{1, 2, "", δ1}, {2, 4, "TT", δ2},
  {2, 3, "FX", δ3}, {3, 4, "TT", δ4}, {3, 5, "FF", δ5}, {4, 5, "", δ6}}
```

```
ItoruP := eCFG[ItoruB, ItoruD, ItoruR, ItoruST]
```

Transformed Program:

```
ItortB := {{"then", 14, 4}}

ItortD :=  {{"IF", {"cond2", 12, 3}, {"cond1", 13, 2}}}

ItortST := {{"start*", 11}, {"stop*", 15}}

ItortR := {{11, 12, "", ρ1}, {12, 14, "TT", ρ2}, {12, 13, "FX", ρ3},
  {13, 14, "TT", ρ4}, {13, 15, "FF", ρ5}, {14, 15, "", ρ6}}

ItortP := eCFG[ItortB, ItortD, ItortR, ItortST]
```

The following relations should be included in the result of *vTransRelGraph*:

$(\rho_1 == \delta_1)$ T1
$(\rho_2 \supseteq \delta_4)$ T2
$(\rho_3 \supseteq \delta_5)$ T3
$(\rho_4 \subseteq \delta_2)$ T4
$(\rho_5 \subseteq \delta_3)$ T5
$(\rho_6 == \delta_6)$ T6
$(\rho_5 == \delta_5)$ T7

■ Condition Swap, Two Conditions with XOR Connective (Csx)

Swapping the conditions of a conditional branch with two conditions connected with XOR. Then and Else Branch are implemented (Bonus example for to demonstrate use of *decisionTrueFalseSet*).

Original Program:

```
CsxuB := {{"then", 4}, {"else", 5}}

CsxuD := {{"decision1", {"cond1", 2}, {"cond2", 3}}}

CsxuST := {{"s", 1}, {"t", 6}}

CsxuR := {{1, 2, "", δ1}, {2, 3, "T1", δ2}, {2, 3, "F2", δ3},
  {2, 3, "T5", δ4}, {2, 3, "F6", δ5}, {3, 4, "F1", δ6}, {3, 4, "T2", δ7},
  {3, 5, "T5", δ8}, {3, 5, "F6", δ9}, {4, 6, "", δ10}, {5, 6, "", δ11}}

CsxuP := eCFG[CsxuB, CsxuD, CsxuR, CsxuST]
```

Transformed Program:

```
CsxtB := {{"then", 14, 4}, {"else", 15, 5}}

CsxtD := {{"decision1", {"cond1", 12, 3}, {"cond2", 13, 2}}}

CsxtST := {{"s", 11, 1}, {"t", 16, 6}}

CsxtR := {{11, 12, "", ρ1}, {12, 13, "T1", ρ2}, {12, 13, "F2", ρ3},
  {12, 13, "T5", ρ4}, {12, 13, "F6", ρ5}, {13, 14, "F1", ρ6}, {13, 14, "T2", ρ7},
  {13, 15, "T5", ρ8}, {13, 15, "F6", ρ9}, {14, 16, "", ρ10}, {15, 16, "", ρ11}}

CsxtP := eCFG[CsxtB, CsxtD, CsxtR, CsxtST]
```



```
CsxHints := {{{δ2}, {ρ7}}, {{ρ7}, {δ2}}, {{δ3}, {ρ6}},
  {{ρ6}, {δ3}}, {{δ4}, {ρ8}}, {{ρ8}, {δ4}}, {{δ5}, {ρ9}},
  {{ρ9}, {δ5}}, {{δ6}, {ρ3}}, {{ρ3}, {δ6}}, {{δ7}, {ρ2}},
  {{ρ2}, {δ7}}, {{δ8}, {ρ4}}, {{ρ4}, {δ8}}, {{δ9}, {ρ5}}, {{ρ5}, {δ9}}}
```

■ Loop Peeling (Lp)

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 and placing them immediatly ahead of the loop (→p. 684). The following graph models the peeling of 1 or 2 iteration(s) with a branch in front of each copy (the increment is thought to be included inside the block).

```
LpuB := {{"loop-body", 2}}

LpuD := {{"while", {"loopcond", 1}}}

LpuR := {{0, 1, "", δ1}, {1, 2, "TT", δ2}, {2, 1, "", δ3}, {1, 3, "FF", δ4}}

LpuS := {"s", 0}

LpuT := {"t", 3}

LpuP := {LpuB, LpuD, LpuR, LpuS, LpuT}
```

Transformed graph, peeling out one copies of the loop-body

```
Lp1tB := {{"lb-copy1", 12, 2}, {"loop-body*", 16, 2}}

Lp1tD := {{"if", {"lc-copy1", 11, 1}}, {"while", {"loopcond", 15, 1}}}

Lp1tR := {{10, 11, "", ρ1}, {11, 12, "TT", ρ2}, {11, 17, "FF-", ρ3},
  {12, 15, "", ρ4}, {15, 16, "TT", ρ8}, {16, 15, "", ρ9}, {15, 17, "FF-", ρ10}}

Lp1tS := {"s", 10, 0}

Lp1tT := {"t", 17, 3}

Lp1tP := {Lp1tB, Lp1tD, Lp1tR, Lp1tS, Lp1tT}
```

Transformed graph, peeling out two copies of the loop-body

```
Lp2tB := {{"lb-copy1", 12, 2}, {"lb-copy2", 14, 2}, {"loop-body*", 16, 2}}

Lp2tD := {{"if", {"lc-copy1", 11, 1}},
  {"if", {"lc-copy2", 13, 1}}, {"while", {"loopcond", 15, 1}}}

Lp2tR := {{10, 11, "", ρ1}, {11, 12, "TT", ρ2}, {11, 17, "FF-", ρ3},
  {12, 13, "", ρ4}, {13, 14, "TT", ρ5}, {13, 17, "FF-", ρ6}, {14, 15, "", ρ7},
  {15, 16, "TT", ρ8}, {16, 15, "", ρ9}, {15, 17, "FF-", ρ10}}

Lp2tS := {"s", 10, 0}

Lp2tT := {"t", 17, 3}

Lp2tP := {Lp2tB, Lp2tD, Lp2tR, Lp2tS, Lp2tT}
```

■ Loop Inversion (Li)

Loop Inversion moves the termination test from beginning of the loop to the end. This modification requires, that the loop body is always executed at least once and that it is safe to do so. In other cases additional test has to be generated in front of the loop to determine wether it is entered. The following model considers the second case and generates a additional decision in front that skips the whole loop, if the loop-termination condition is true at beginning of the loop.

```
LiuB := {{"loop-body", 3}}

LiuD := { {"while", {"loopcond", 2}}}

LiuR := {{1, 2, "", δ1}, {2, 3, "TT", δ2}, {3, 2, "", δ3}, {2, 4, "FF", δ4}}

LiuS := {"s", 1}

LiuT := {"t", 4}

LiuP := {LiuB, LiuD, LiuR, LiuS, LiuT}
```

Now the definition of the modified graph. The fork 12-15 is marked with "-", because although the decision made at node 12 is the same as the loop control the *False* branch only covers a subset of input valuations compared with 2, where the *False* branch is the main exit of the loop that comprises all input valuations.

```
LitB := {{"loop-body", 13, 3}}

LitD := {{"if", {"c1", 12, 2}}, {"while", {"c2", 14, 2}}}

LitR := {{11, 12, "", ρ1}, {12, 13, "TT", ρ2}, {12, 15, "FF-", ρ3},
  {13, 14, "", ρ4}, {14, 13, "TT", ρ5}, {14, 15, "FF-", ρ6}}

LitS := {"s", 11, 1}

LitT := {"t", 15, 4}

LitP := {LitB, LitD, LitR, LitS, LitT}
```

■ Loop Fusion (Lf)

Loop Fusion takes two adjacent loops that have the same iteration-space traversal and combines their bodies into a single loop.

```
LfuB := {{"body1", 2}, {"body2", 4}}

LfuD := {{"while1", {"cond", 1}}, {"while2", {"cond", 3}}}

LfuR := {{0, 1, "", δ1}, {1, 2, "TT", δ2}, {2, 1, "", δ3},
  {1, 3, "FF", δ4}, {3, 4, "TT", δ5}, {4, 3, "", δ6}, {3, 5, "FF", δ7}}

LfuS := {"s", 0}

LfuT := {"t", 5}

LfuP := {LfuB, LfuD, LfuR, LfuS, LfuT}
```

Now the optimized program:


```
LftB := {{"body1", 12, 2}, {"body2", 13, 3}}

LftD := {{"while", {"cond", 11, 1}}}

LftR := {{10, 11, "", ρ1}, {11, 12, "TT", ρ2},
  {12, 13, "", ρ3}, {13, 11, "", ρ4}, {11, 14, "FF", ρ5}}

LftS := {"s", 10, 0}

LftT := {"t", 14, 5}

LftP := {LftB, LftD, LftR, LftS, LftT}
```

■ Loop Interchange (Lic)

Loop Interchange (or more general *Loop Permuation*) swaps the order, nested loops are processed. This use case shows a model for a double nested loop where the inner and outer loop are swaped. Each loop decision consists of two conditions.

```
LicuB := {{"body", 6}}

LicuD := {{"loop1", {"cond11", 2}, {"cond12", 3}},
  {"loop2", {"cond21", 4}, {"cond22", 5}}}

LicuR := {{1, 2, "", δ1}, {2, 3, "TX", δ2}, {2, 3, "FF", δ3}, {3, 7, "TF", δ5},
  {3, 7, "FF", δ6}, {3, 4, "TT", δ4}, {4, 5, "TX", δ7}, {4, 5, "FF", δ8},
  {5, 6, "TT", δ9}, {5, 2, "TF", δ11}, {5, 2, "FF", δ12}, {6, 4, "", δ10}}

LicuST := {{"s", 1}, {"t", 7}}

LicuP := eCFG[LicuB, LicuD, LicuR, LicuST]
```

Now the optimized program:

```
LictB := {{"body", 16, 6}}

LictD := {{"loop1*", {"cond21", 12, 4}, {"cond22", 13, 5, "X="}},
  {"loop2*", {"cond11", 14, 2}, {"cond12", 15, 3}}}

LictR := {{11, 12, "", ρ1}, {12, 13, "TX", ρ2}, {12, 13, "FF", ρ3},
  {13, 17, "TF", ρ5}, {13, 17, "FF", ρ6}, {13, 14, "TT", ρ4},
  {14, 15, "TX", ρ7}, {14, 15, "FF", ρ8}, {15, 16, "TT", ρ9},
  {15, 12, "TF", ρ11}, {15, 12, "FF", ρ12}, {16, 14, "", ρ10}}

LictST := {{"s'", 11, 1}, {"t'", 17, 7}}

LictP := eCFG[LictB, LictD, LictR, LictST]
```

Theory for transformation conditions: A Loop Interchange is valid, if it is true for all input-valuations, that the same input valuation triggers the same number of iterations of the statements inside the innermost loop in the non-transformed program as well as in the transformed program. Therefore, the input-valuations entering the innermost loop must be the same and there could be no additional input valuations inside the innermost loop. If the number of loop iterations is not dependent on the input-valuations, this is obvious. If the loop decision are dependent on the input-valuations then, assuming that the loop-decision are independent from each other, each input-valuations entering the innermost loop must evaluate each loop-decision to *True* at least once and therefore, because the decisions are the same, this must be true also in the transformed program. This implies T1 ($\rho_9 == \delta_9$). All input valuations entering the loop structure must leave it through the fork 2→7 resp. 12→17 which implies $\delta_5 \bigcup \delta_6 == \rho_5 \bigcup \rho_6$. For all other input-valuations, not entering the innermost loop, there is no statement possible, where they will be rejected. This can happen either by the first or the second loop decision

(please be aware, that the loop conditions can be dependent and therefore swapping the loop decisions can cause a shift of input-valuations with outgoing *False* from one decision to another decision). Therefore the relation for the *True*-fork of decision {12,13} can be $\subseteq$, == or $\supseteq$. Therefore we define the functional relation *X* in node 13 to avoid creation of a *True* relation.

Following transition relations seems to be the most importent and should be found by *vTransRelGraph* or added manually.

$\rho_1 == \delta_1$
$\rho_9 == \delta_9$ (T1)

T1 describes the equality of the input-valuations in the innermost loop.

```
LicT1 := {{{δ9}, {ρ9}}, {{ρ9}, {δ9}}}
```

■ Loop Unrolling (Lur)

Loop Unrolling replaces the body of a loop by several copies of the body and adjusts the loop-control code accordingly. The number of copies *K* (*K*=2 in this example) is called the *unrolling factor*. The original loop is called the *rolled loop*.

```
LuruB := {{"body", 3}}

LuruD := {{"loop", {"cond", 2}}}

LuruR := {{1, 2, "", δ1}, {2, 3, "TT", δ2}, {3, 2, "", δ3}, {2, 4, "FF", δ4}}

LuruST := {{"s", 1}, {"t", 4}}

LuruP := eCFG[LuruB, LuruD, LuruR, LuruST]
```

Now the optimized program where node 3 is duplicated with unrolling factor *K*=2 (nodes 13,14) and a copy of the rolled loop (Decision 15, body 16) to handle iterations not a multiple of *K*:

```
LurtB := {{"unrolledBody1", 13, 3}, {"unrolledBody2", 14, 3}, {"body", 16, 3}}

LurtD := {{"unrolledLoop", {"cond", 12, 2}}, {"loop", {"cond", 15, 2}}}

LurtR := {{11, 12, "", ρ1}, {12, 13, "TT-", ρ2}, {13, 14, "", ρ3}, {14, 12, "", ρ4},
  {12, 15, "FF", ρ5}, {15, 16, "TT-", ρ6}, {16, 15, "", ρ7}, {15, 17, "FF", ρ8}}

LurtST := {{"s", 11, 1}, {"t", 17, 4}}

LurtP := eCFG[LurtB, LurtD, LurtR, LurtST]
```

Following Transition Relations should be found by *vTransRelGraph* for Loop Unrolling:

$\rho_1 == \delta_1$
$\rho_2 == \delta_2$
$\rho_3 == \delta_5$
$\rho_5 == \delta_7$

```
LurHints := {}
```


■ Strip Mining, 2 Conditions Per Decision (Sm)

Strip Mining divides a loop into a series of loops operating on strips of the original one as in the following example:

Remark: To get the correct result, the 2 Condition per Decision Version must be used, because decision coverage is preserved and a single condition always goes with the decision.

The following CFG represents the untransformed program (a):

```
SmuB := {{"statement-block", 4}}

SmuD := {{"i-loop", {"cond-i1", 2}, {"cond-i2", 3}}}

SmuR := {{1, 2, "", δ1}, {2, 3, "TX", δ2}, {2, 3, "FF", δ3},
  {3, 4, "TT", δ4}, {3, 5, "TF", δ5}, {3, 5, "FF", δ6}, {4, 2, "", δ7}}

SmuST := {{"s", 1}, {"t", 5}}

SmuP := eCFG[SmuB, SmuD, SmuR, SmuST]
```

Strip Mining creates a 2 level deep loop nest (b) by splitting the *i*-loop:

```
SmtB := {{"statement-block", 16, 4}}

SmtD := {{"i-loop", {"cond-i1", 12, 2}, {"cond-i2", 13, 3}},
  {"i1-loop", {"cond-i1-1", 14, 2}, {"cond-i1-2", 15, 3}}}

SmtR := {{11, 12, "", τ1}, {12, 13, "TX", τ2}, {12, 13, "FF", τ3}, {13, 14, "TT", τ4},
  {13, 17, "TF", τ5}, {13, 17, "FF", τ6}, {14, 15, "TX", τ7}, {14, 15, "FF", τ8},
  {15, 16, "TT", τ9}, {15, 12, "TF", τ10}, {15, 12, "FF", τ11}, {16, 14, "", τ12}}

SmtST := {{"s", 11, 1}, {"t", 17, 5}}

SmtP := eCFG[SmtB, SmtD, SmtR, SmtST]
```

Following transition relations are essential to analyse transformations:

$\delta_1 == \tau_1$
$\delta_4 == \tau_9$
$\delta_4 \subseteq \tau_4$

```
SmHints := {{{δ4}, {τ9}}, {{τ9}, {δ4}}, (*{{δ4},{τ4}},*){{τ4}, {δ4}}}
```

■ Loop Tiling (Lt)

Tiling (other names: *Blocking*, *Strip Mine and Interchange*, *Unroll and Jam*) of a single loop replaces it by a pair of loops with the inner one (called the *tile loop*) having an incremental like the original loop and the outer one having an incremental equal to *ub-lb+1*, where *lb* and *ub* are the lower and upper bounds of the inner loop. *Tiling* a loop nest of depth *n* may increases the depth of the loop nest anywhere from *n+1* up to *2n*, depending on how many of the loops are tiled. Tiling also interchanges the loops beginning from the tiled one inward to make the tile loops the innermost one in the loop nest. The number of iterations of the tile loop is called the *tile size*.

The following CFG represents the untransformed program (a):

```
LtuB := {{"statement-block", 6}}
```

```
LtuD := {{"i-loop", {"cond-i1", 2}, {"cond-i2", 3}},
  {"j-loop", {"cond-j1", 4}, {"cond-j2", 5}}}

LtuR := {{1, 2, "", δ1}, {2, 3, "TX", δ2}, {2, 3, "FF", δ3}, {3, 4, "TT", δ4},
  {3, 7, "TF", δ5}, {3, 7, "FF", δ6}, {4, 5, "TX", δ7}, {4, 5, "FF", δ8},
  {5, 6, "TT", δ9}, {5, 2, "TF", δ10}, {5, 2, "FF", δ11}, {6, 4, "", δ12}}

LtuST := {{"s", 1}, {"t", 7}}

LtuP := eCFG[LtuB, LtuD, LtuR, LtuST]
```

First we create a tile-loop for both loops resulting in a 4 level deep loop nest. In the second step we do an interchange of the two loops in the middle to bring it inwards.

```
LttB := {{"statement-block", 20, 6}}

LttD := {{"i-loop", {"cond-i1", 12, 2}, {"cond-i2", 13, 3}},
  {"j-loop", {"cond-j1", 14, 4}, {"cond-j2", 15, 5}},
  {"i1-loop", {"cond-i1-1", 16, 2}, {"cond-i1-2", 17, 3}},
  {"j1-loop", {"cond-j1-1", 18, 4}, {"cond-j1-2", 19, 5}}}

LttR := {{11, 12, "", ρ1}, {12, 13, "TX", ρ2}, {12, 13, "FF", ρ3},
  {13, 14, "TT", ρ4}, {13, 21, "TF", ρ5}, {13, 21, "FF", ρ6}, {14, 15, "TX", ρ7},
  {14, 15, "FF", ρ8}, {15, 16, "TT", ρ9}, {15, 12, "TF", ρ10}, {15, 12, "FF", ρ11},
  {16, 17, "TX", ρ12}, {16, 17, "FF", ρ13}, {17, 18, "TT", ρ14}, {17, 14, "TF", ρ15},
  {17, 14, "FF", ρ16}, {18, 19, "TX", ρ17}, {18, 19, "FF", ρ18},
  {19, 20, "TT", ρ19}, {19, 16, "TF", ρ20}, {19, 16, "FF", ρ21}, {20, 18, "", ρ22}}

LttST := {{"s", 11, 1}, {"t", 21, 7}}

LttP := eCFG[LttB, LttD, LttR, LttST]
```

Following Transition Relations should be found by *vTransRelGraph* for Loop Tiling:

$$\rho_1 == \delta_1$$
$$\rho_2 == \delta_2$$
$$\rho_3 == \delta_5$$
$$\rho_5 == \delta_7$$
$$\delta_9 == \rho_{19}$$

```
LtHints := {{{δ9}, {ρ19}}, {{ρ19}, {δ9}}}
```

▪ Unswitching with 2 Conditions per Decision (Usw2)

...is a loop optimization that pulls out a branch which does not depend on the iteration variable. E.g. let's assume, the loop iteration variable is *i* but the branch only depends on the value of a variable *k*:

```
Usw2uB := {{"then-statement1", 6}, {"else-statement2", 7}, {"join-statement", 8}}

Usw2uD := {{"i-loop", {"i-cond1", 2}, {"i-cond2", 3}},
  {"if", {"k-cond1", 4}, {"k-cond2", 5}}}

Usw2uR := {{1, 2, "", δ1}, {2, 3, "TX", δ2},
  {2, 3, "FF", δ3}, {3, 4, "TT", δ4}, {3, 8, "TF", δ5}, {3, 8, "FF", δ6},
  {4, 5, "TX", δ7}, {4, 5, "FF", δ8}, {5, 6, "TT", δ9}, {5, 7, "TF", δ10},
  {5, 7, "FF", δ11}, {6, 8, "", δ12}, {7, 8, "", δ13}, {8, 2, "", δ14}}

Usw2uST := {{"s", 1}, {"t", 8}}
```



```
Usw2uP := eCFG[Usw2uB, Usw2uD, Usw2uR, Usw2uST]
```

Now the optimized program:

```
Usw2tB := {{"then-statement1", 16, 6}, {"else-statement2", 19, 7}}

Usw2tD := {{"i-loop-then", {"i-cond1", 14, 2}, {"i-cond2", 15, 3}},
  {"i-loop-else", {"i-cond1", 17, 2}, {"i-cond2", 18, 3}},
  {"if", {"k-cond1", 12, 4}, {"k-cond2", 13, 5}}}

Usw2tR := {{11, 12, "", ρ1}, {12, 13, "TX", ρ2}, {12, 13, "FF", ρ3},
  {13, 14, "TT", ρ4}, {13, 17, "TF", ρ5}, {13, 17, "FF", ρ6}, {14, 15, "TX", ρ7},
  {14, 15, "FF", ρ8}, {15, 16, "TT", ρ9}, {15, 20, "TF", ρ10},
  {15, 20, "FF", ρ11}, {16, 14, "", ρ12}, {17, 18, "TX", ρ13}, {17, 18, "FF", ρ14},
  {18, 19, "TT", ρ15}, {18, 20, "TF", ρ16}, {18, 20, "FF", ρ17}, {19, 17, "", ρ18}}

Usw2tST := {{"s*", 11, 1}, {"t*", 20, 8}}

Usw2tP := eCFG[Usw2tB, Usw2tD, Usw2tR, Usw2tST]
```

Theory for transformation conditions: With the same arguments than above the basic transformation conditions are as follows:

```
Usw2Hints := {{{δ12}, {δ9}}, {{δ9}, {δ12}}, {{δ13}, {δ15}}, {{δ15}, {δ13}}}
```

▪ Software Pipelining (Swpl)

Software Pipelining (also called *kernel scheduling*) is a preperation step to make use of parallelism across loop iterations. It reorganizes a loop into three components: (1) a *kernel* including the code that must be executed on every cycle of the loop, once it has reached a steady state, (2) a *prolog*, which includes the code that must be executed before the steady state can be reached and (3) an *epilog*, which must be executed to finish the loop once the kernel can no longer been executed.

The goal of kernel scheduling is to focus on temporal movement of instructions through loop iterations rather than on spatial movement within a single loop iteration. Critical instructions whose results are needed early are moved to earlier loop iterations, so that their results become available within the current iteration just as they are needed. Similarly, instructions at the tail end of the critical path are moved to future iterations so as to shorten completion of the current iteration. In other words, the body of one loop iteration is piplinded across multiple iterations in order to take fullest advantage of available resources within one iteration.

The following CFG represents the non-transformed program (a):

```
SwpuB := {{"statement1(i)", 3}, {"statement2(i)", 4}}

SwpuD := {{"loop-i", {"loopcond", 2}}}

SwpuR := {{1, 2, "", δ1}, {2, 3, "TT", δ2},
  {2, 5, "FF", δ3}, {3, 4, "", δ4}, {4, 2, "", δ5}}

SwpuST := {{"s", 1}, {"t", 5}}

SwpuP := eCFG[SwpuB, SwpuD, SwpuR, SwpuST]
```

Transformed graph, peeling out prolog-/epilog-copies of the loop-body:

```
SwptB := {{"statement1(1)", 12, 3}, {"statement2(n)", 16, 4},
  {"statement1(i)", 15, 3}, {"statement2(i-1)", 14, 4}}
```

```
SwptD := {{"loop-i", {"loopcond", 13, 2}}}

SwptR := {{11, 12, "", ρ1}, {12, 13, "", ρ2}, {13, 14, "TT-", ρ3},
  {14, 15, "", ρ4}, {15, 13, "", ρ5}, {13, 16, "FF", ρ6}, {16, 17, "", ρ7}}

SwptST := {{"s", 11, 1}, {"t", 17, 5}}

SwptP := eCFG[SwptB, SwptD, SwptR, SwptST]
```

Transition Relations for Software Pipelining:

■ Branch Optimization (Bo)

Branch Optimization redirects a jump to a unconditional jump immediately to the target of the unconditional jump.

```
BouB := {{"Jump-Statement", 2}, {"Jump-Unconditional", 3}, {"Branch-Target", 4}}

BouD := {}

BouST := {{"s", 1}, {"t", 5}}

BouR := {{1, 2, "", δ1}, {2, 3, "", δ2}, {3, 4, "", δ3}, {4, 5, "", δ4}}

BouP := eCFG[BouB, BouD, BouR, BouST]
```

After redirecting the jump, the unconditional jum instruction 13 is unconnected on the input side.

```
BotB := {{"Jump-Statement", 12, 2},
  {"Jump-Unconditional", 13, 3}, {"Branch-Target", 14, 4}}

BotD :=  {}

BotST := {{"s", 11, 1}, {"t", 15, 5}}

BotR := {{11, 12, "", ρ1}, {12, 14, "", ρ2}, {13, 14, "", ρ3}, {14, 15, "", ρ4}}

BotP := eCFG[BotB, BotD, BotR, BotST]
```

■ End Of Use Cases

Automatic Preservation Analysis

This section provides complete functions for analysis of all coverage preservation condition for one use case as well as a matrix function to produce a overall matrix scheme for als preservation criteria for different usecases.

■ cUseCaseAnalysis

First we start with a analysis function for one usecase. It returns a line vector including usecase-name and entries for SC w/o C. (Statement Coverage without including conditions), SC w. C. (Statement Coverage treating conditions as statements), CC (Condition Coverage), DC (Decision Coverage), MCDC (Modified Condition Decision Coverage) and PC (Path Coverage) in this order.


```
cUseCaseAnalysis[name_, p1_, p2_, hints_, pathBased_] :=
 Module[{startTime, vfr, sc1, sc2, cc, dc, mcdc, pc},
  startTime = SessionTime[];
  If[MemberQ[outputSet, 0],
   If[pathBased,
    Print[">>> Processing Use-Case \"",
     name, "\", PATH-based valuation-relations analysis."],
    (* else *)
    Print[">>> Processing Use-Case \"",
     name, "\", SIMPLE valuation-relations analysis."]
   ];
   If[hints ≠ {},
    Print["    Hints:"];
    vPrintRelationSet[hints, "       "]
   ]
  ];

  If[MemberQ[outputSet, 1],
   Print["Program P1:"];
   gDrawCfg[p1];
   Print["B(P1): ", MatrixForm[gB[p1]], "  D(P1): ",
    MatrixForm[gD[p1]], "   ST(P1):", MatrixForm[gST[p1]]];
   Print["Program P2:"];
   gDrawCfg[p2];
   Print["B(P2): ", MatrixForm[gB[p2]], "  D(P2): ",
    MatrixForm[gD[p2]], "   ST(P2):", MatrixForm[gST[p2]]]
  ];

  vfr = vTransRelGraph[p1, p2, hints, pathBased];
  If[MemberQ[outputSet, 3],
   Print["Input-Valuation Relations obtained:"];
   vPrintRelationSet[vfr, "   "]
  ];

  sc1 = cSCPres[p1, p2, vfr];
  sc2 = cSCPres[eCFG[gB[p1] ⋃ gC[p1], gD[p1], gR[p1], gST[p1]],
    eCFG[gB[p2] ⋃ gC[p2], gD[p2], gR[p2], gST[p2]], vfr];
  cc = cCCPres[p1, p2, vfr];
  dc = cDCPres[p1, p2, vfr];
  mcdc = cMCDCPres[p1, p2, vfr];
  pc = cPCPres[p1, p2, vfr];

  If[MemberQ[outputSet, 0],
   Print["<<< Finished \"", name,
    "\", time elapsed: ", SessionTime[] - startTime, " sec."]
  ];

  {name, sc1, sc2, cc, dc, mcdc, pc}
 ] (* UseCaseAnalysis *)
```

■ cPresAnalysis

cPresAnalysis just calls *UseCaseAnalysis* with different use cases. To display the result as matrix call the function with *MatrixForm[cPresAnalysis[]]*. Otherwise the result is presented in a "list of list" form. The argument *outLevel* is a value that maps to a temporary global *outputSet* used during runtime of the function. For convenience integer numbers from 0 to 5 are used. Values of 0 to 5 establish a certain temporary *outputSet* (→ *outLevelMap* in the function body). A value of -1 makes the output silent. A higher number produces more output than a lower number. 0 produces more or less only progress messages, 1 produces mainly output related to the coverage preservation theory. 2,3,4,5 go more deep into the interals of the perservation proofs.

```
cPresAnalysis[outLevel_] :=
 Module[{saveOutSet, overallResult, outLevelMap},
  overallResult = {{"Use Case", "SC w/o C.", "SC w. C.",
     "CC", "DC", "MCDC", "PC"}, {"--------", "---------",
     "--------", "-----", "-----", "-----", "-----"}};
  outLevelMap = {
    {0},                        (* 0 *)
    {0, 2, 3},                  (* 1 *)
    {0, 2, 3, 4, 5},            (* 2 *)
    {0, 2, 3, 4, 5, 6},         (* 3 *)
    {0, 1, 2, 3, 4, 5, 6},      (* 4 *)
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  (* 5 *)
   };

  saveOutSet = outputSet;
  If[outLevel < 0,
   outputSet = {},
   outputSet = outLevelMap[[Min[outLevel + 1, 5]]]
  ];

  (* Useless Code Elimination
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Useless Code Elimination", UceuP, UcetP, {}, False]];

  (* Two Condition If-Then-Else Branch with Full Condition-Evaluation
   *)
  overallResult = Append[overallResult, cUseCaseAnalysis[
     "Full-Evaluated If-Then-Else", TcfruP, TcfrtP, {}, False]];

  (* Two Condition If-Then-Else Branch with Shortcut Evaluation
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Shortcut If-Then-Else", TcsruP, TcsrtP, {}, False]];

  (* Two Condition Branch (AND) with empty ELSE and Shortcut Evaluation
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Empty Else Shortcut AND", ItaruP, ItartP, {}, False]];

  (* Two Condition Branch (OR) with empty ELSE and Shortcut Evaluation
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Empty Else Shortcut OR", ItoruP, ItortP, {}, False]];

  (* Loop Peeling (peeling out 1 Step)
   *)
  overallResult = Append[overallResult,
```



```
    cUseCaseAnalysis["Loop Peeling (k=1)", LpuP, LpltP, {}, False]];

  (* Loop Inversion
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Loop Inversion", LiuP, LitP, {}, False]];

  (* Loop Fusion
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Loop Fusion", LfuP, LftP, {}, False]];

  (* Loop Interchange
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Loop Interchange", LicuP, LictP, LicT1, True]];

  (* Loop Unrolling
   *)
  overallResult = Append[overallResult, cUseCaseAnalysis[
     "Loop Unrolling (k=2)", LuruP, LurtP, LurHints, True]];

  (* Strip Mining
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Strip Mining", SmuP, SmtP, SmHints, False]];

  (* Loop Tiling
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Loop Tiling", LtuP, LttP, LtHints, True]];

  (* Unswitching
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Loop Unswitching", Usw2uP, Usw2tP, Usw2Hints, True]];

  (* Software Pipelining
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Software Pipelining", SwpuP, SwptP, {}, False]];

  (* Branch Optimization
   *)
  overallResult = Append[overallResult,
    cUseCaseAnalysis["Branch Optimization", BouP, BotP, {}, True]];

  outputSet = saveOutSet;

  overallResult
 ] (* Module *)
```

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Co., 3rd edition.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. PRENTICE HALL, 1993.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.

[4] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[5] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Elsevier Academic Press, 2002.

[6] Zena M. Ariola and Arvind. A syntactic approach to program transformations, 1991.

[7] Uwe Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 121–135, London, UK, 1996. Springer-Verlag.

[8] Andre Baresel, Mirko Conrad, Sadegh Sadeghipour, and Joachim Wegener. The interplay between model coverage and code coverage.

[9] P. V. Bhansali. The mcdc paradox. *SIGSOFT Softw. Eng. Notes*, 32(3):1–4, 2007.

[10] Vittorio Cortellessa Bojan, Bojan Cukic, Diego Del Gobbo, Ali Mili, Marcello Napolitano, Mark Shereshevsky, and Harjinder S. Certifying adaptive flight control software. In *In Proceedings, ISACC 2000: The Software Risk Management Conference*, 2000.

[11] Bela Bollobas. *Modern Graph Theory*. Springer, 1998.

[12] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. In *Software Engineering Journal*, volume 9, pages 193–200. IEEE Xplore, September 1994.

[13] Bruce A. Cota, Douglas G. Fritz, and Robert G. Sargent. Control flow graphs as a representation language. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 555–559, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[14] Oliver Deiser. *Einführung in die Mengenlehre*. Springer, 2nd edition, 2004.

[15] George Devaraj, Mats P. E. Heimdahl, and Donglin Liang. Condition based coverage criteria: To use or not to use, that is the question.

[16] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[17] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*. Springer-Verlag Berlin-Heidelberg, 1995.

[18] Sebastian Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. *Software Maintenance, IEEE International Conference on*, 0:170, 2001.

[19] Robert W. Floyd. Assigning meanings to programs. In *Proc. of AMS Symposia in Applied Mathematics*, pages 19–32, 1967.

[20] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19:202–213, 1993.

[21] W. Gifford. Structural coverage analysis method. In *Digital Avionics Systems Conference, 1996., 15th AIAA/IEEE*, page 43, oct 1996.

[22] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications*. Chapman and Hall/CRC, 2nd edition, 2006.

[23] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.

[24] Kelly Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage, 2001.

[25] M.P.E. Heimdahl, M.W. Whalen, A. Rajan, and M. Staats. On mc/dc and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 5.B.3–1 –5.B.3–13, oct. 2008.

[26] Nevin Heintze. Control-flow analysis and type systems, 1995.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[28] Michael Huth and Mark Ryan, editors. *Logic in Computer Science*. Cambridge University Press, 2nd edition, 2004.

[29] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Vienna University of Technology, 2003.

[30] Raimund Kirner. SCCP/x - a compilation profile to support testing and verification of optimized code. In *Proc. ACM Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*, pages 38–42, Salzburg, Austria, Sep./Oct. 2007.

[31] Raimund Kirner. Towards preserving model coverage and structural code coverage. *EURASIP J. Embedded Syst.*, 2009:1–16, 2009.

[32] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, New York, NY, USA, 1981. ACM.

[33] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–294. ACM Press, 2002.

[34] David Lacey and Oege De Moor. Imperative program transformation by rewriting. In *in: Proc. Compiler Construction 2001, LNCS 2027, 2001*, pages 52–68. Springer Verlag, 2001.

[35] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation Atlanta, Georgia*, pages 318–328, 1988.

[36] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 220–231. ACM Press, 2003.

[37] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969.

[38] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[39] Randall Mercer. The convex fortran 5.0 compiler. In *Proceedings of the 3rd International Conference on Supercomputing, Vol. 2*, 1988.

[40] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[41] Glenford J. Myers. *The Art of Software Testing*. Wiley and Son Inc., 2nd edition, 2004.

[42] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14:868–874, 1988.

[43] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[44] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46 –57, 31 1977-nov. 2 1977.

[45] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings. Eighth Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems, 2001. ECBS 2001.*, pages 83 –91, 2001.

[46] Orna Raz, Moshe Klausner, Nitzan Peleg, Gadi Haber, Eitan Farchi, Shachar Fienblit, Yakov S. Filiarsky, Shay Gammer, and Sergey Novikov. The advantages of post-link code coverage. In Karen Yorav, editor, *Haifa Verification Conference*, volume 4899 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2007.

[47] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[48] Olin Shivers. Control flow analysis in scheme, 1988.

[49] Steven S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley Publishing Co., 1990.

[50] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the z notation. In *Proceedings of COMPSAC 2001: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. Society Press, 2001.

[51] Eelco Visser. A survey of rewriting strategies in program transformation systems. In *In Workshop on Reduction Strategies in Rewriting and Programming (WRS'01), volume 57 of Electronic Notes in Theoretical Computer Science*, page 2001. Elsevier Science Publishers, 2003.

[52] M.D. Weiser, J.D. Gannon, and P.R. McMullin. Comparison of structural test coverage metrics. *IEEE Software*, 2:80–85, 1985.

[53] Paul R. Wellin, Richard J. Gaylord, and Samuel N. Kamin. *An Introduction to Programming with Mathematica*. Cambridge University Press, 3rd edition, 2008.

[54] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008.

[55] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM.

[56] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.

[57] Wolfram research inc., online tutorial: General graph drawing. http://reference.wolfram.com/mathematica/tutorial/GraphDrawing.html, 2010.

[58] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 4th edition, 1999.

[59] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng.*, 6(3):278–286, 1980.

[60] Michael Zolda, Sven Bünte, and Raimund Kirner. Towards adaptable control flow segmentation for measurement-based execution time analysis. In *Proc. 17th International Conference on Real-Time and Network Systems (RTNS)*, Paris, France, Oct. 2009.

[61] Lenore Zuck, Amir Pnueli, Yi Fang, Benjamin Goldberg, and Ying Hu. Translation and run-time validation of optimized code. *Electronic Notes in Theoretical Computer Science*, 70(4):179 – 200, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).