Degree Project in Technology

Second cycle, 30 credits

The Logic of Glass Box Test Coverage

A formal model for coverage through weakest preconditions

HUGO TRICOT

The Logic of Glass Box Test Coverage

A formal model for coverage through weakest preconditions

HUGO TRICOT

Date: March 10, 2025

Supervisor: Karl MEINKE
Examiner: Roberto GUANCIALE
School of Electrical Engineering and Computer Science
Swedish title: Logiken bakom testtäckning i glasbox
Swedish subtitle: En formell modell för täckning genom svagaste förhandsvillkor

© 2025 Hugo TRICOT

Abstract

Testing is an essential activity of software development, and despite the vast use of testing by the industry, little formal reasoning can be found in the scientific literature of the field. The project presents an extensively formal approach to glass box testing from the underlying language to the graph model of a program, its execution, and reasoning on preconditions for its paths. In this paper, programs are modelled as executable Control Flow Graphs and test requirements are paths of the graphs.

We study a logic to infer the weakest precondition of a given path of the graph. Weakest preconditions let us reason on relations between test requirements, which in turn let us optimise the test requirements set. The model uses a minimal language with the integer data type and first-order logic. This language can be extended as defined in this work to suit the need of the reader.

Several metatheorems are proved for any language extending the minimal language we provide, including soundness, completeness under the restrictions to obtain a weakest precondition formula, and undecidability.

Keywords

Software testing, Glass box testing, Test requirement redundancy, Mathematical models, Proof systems, Control-flow graphs

Sammanfattning

Testning är en viktig aktivitet inom programvaruutveckling, och trots den omfattande användningen av testning inom industrin finns det få formella resonemang i den vetenskapliga litteraturen inom området. Projektet presenterar ett omfattande formellt tillvägagångssätt för glasbox-testning från det underliggande språket till grafmodellen för ett program, dess exekvering och resonemang om förutsättningar för dess vägar. I detta dokument modelleras program som exekverbara kontrollflödesgrafer och testkrav är banor i graferna.

Vi studerar en logik för att härleda det svagaste förhandsvillkoret för en given väg i grafen. Svagaste förhandsvillkor låter oss resonera om relationer mellan testkrav, vilket i sin tur låter oss optimera testkravsuppsättningen. Modellen använder ett minimalt språk med datatypen heltal och första ordningens logik. Detta språk kan utökas enligt definitionen i detta arbete för att passa läsarens behov.

Flera metateorem bevisas för alla språk som utvidgar det minimala språk vi tillhandahåller, inklusive sundhet, fullständighet under restriktionerna för att få en svagaste förhandsvillkorformel och oavgörbarhet.

Nyckelord

Programvara testning, Glasbox-testning, Testkrav redundans, Matematiska modeller, Bevissystem, Kontrollflödesdiagram

Résumé

L'activité de test est essentielle au développement logiciel, et malgré la vaste utilisation du test dans l'industrie, la littérature du domaine comporte peu de raisonnement formel. Le projet présente une approche extensivement formelle du test structurel, du langage sous-jacent au graphe modélisant le programme, son exécution et le raisonnement sur les préconditions des chemins dudit graphe. Dans ce document, les programmes sont modélisés en graphes de flux de contrôle exécutables et les exigences de tests sont des chemins desdits graphes.

Nous étudions une logique pour déduire la précondition minimale d'un chemin donné du graphe. Les préconditions minimales nous permettent de raisonner sur les relations entre exigences de tests. Le modèle utilise un langage minimal incluant le type des entiers et les formules de premier ordre. Plusieurs méta-théorèmes sont prouvés pour tout langage étendant le langage minimal fourni, dont la cohérence, la complétude sous les restrictions pour obtenir des préconditions minimales, et l'indécidabilité.

Mots-clés

Test logiciel, Test structurel, Redondance d'exigences de test, Modèles mathématiques, Systèmes de preuve, Graphes de flux de contrôle

Acknowledgments

I would like to thank Prof. Karl Meinke for his invaluable help and insights during this work. I would also like to thank Dr. Roberto Guanciale for the time he spent on this thesis. Finally, I would like to thank Prof. Wolfgang Ahrendt for pointing me to the KeY project's book.

Stockholm, March 2025
Hugo TRICOT

Contents

1 Introduction 1

1.1 Problem statement . . . 1

1.2 Motivations . . . 1

1.3 Methods and concepts . . . 2

1.4 Goals and hypothesis . . . 4

1.5 Delimitations . . . 4

1.6 Structure of the thesis . . . 5

2 Background and literature survey 7

2.1 Mathematical concepts . . . 7

2.1.1 Control flow graphs . . . 7

2.1.2 Structures and data types . . . 9

2.1.3 Proof systems . . . 11

2.2 Testing . . . 12

2.2.1 Test coverage . . . 12

2.2.2 Redundancy reduction . . . 14

2.2.3 Weakest preconditions . . . 17

2.3 Methodology and software verification . . . 18

2.4 Summary . . . 20

3 Methodology 21

3.1 Software errors . . . 21

3.2 Signatures and structures . . . 22

3.2.1 Languages foundations . . . 22

3.2.2 Assignments . . . 24

3.2.3 Terms, formulas and evaluations . . . 25

3.2.4 Required signature for graphs . . . 28

3.3 Control Flow Graphs . . . 30

3.3.1 Well-formed graphs . . . 30

3.3.2 Paths and matching 34
3.3.3 Graph execution 35
3.3.4 Remark on graph creation 38
3.4 Testing requirements 39
3.4.1 Requirements and path satisfiability 39
3.4.2 Testing concepts 43
3.5 Inference rules 45
3.5.1 Precondition calculus 46
3.5.2 Proof trees and provable statements 48
3.5.3 Soundness 50
3.5.4 Completeness 53
3.6 Weakest precondition, coverage and calculus properties 54
3.6.1 Coverage and relations 54
3.6.2 Precondition calculus and weakest preconditions . . . 57
3.6.3 Achieved completeness 64
3.7 Calculus comparison 65
3.8 Summary 67

4 Discussions and future work **71**
4.1 Limitations 71
4.1.1 Incompleteness 71
4.1.2 Weakest preconditions 72
4.1.3 Error handling 73
4.2 Future work 74
4.2.1 Postcondition calculus 74
4.2.2 Stricter structure flow graphs 75
4.2.3 Automation of test case generation 76
4.3 Relation to other problems 76
4.4 Summary 77

5 Conclusion **79**
5.1 Conclusions 79
5.2 Ethics and sustainability 80

References **83**

List of Figures

2.1 A simple and arbitrary CFG 8

3.1 Program *myPoly* and its Control Flow Graph (CFG) 32
3.2 Program *isEven* and its CFG 44
3.3 Graphical representation of weakest preconditions 55
3.4 Loop impact on the weakest precondition 61

List of Tables

3.1 Summary of important notation 68

List of acronyms and abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
CFG	Control Flow Graph
DBPC	Disjunctive Branch Precondition
DILP	Degraded Integer Linear Programming
EC	Edge Coverage
F-DBPC	Full Disjunctive Branch Precondition
FOL	First-Order Logic
GA	Genetic Algorithm
GSA	Gravitational Search Algorithm
ILP	Integer Linear Programming
MC/DC	Modified Condition/Decision Coverage
MUT	Method Under Test
NC	Node Coverage
PC	Predicate Coverage
SDG	Sustainable Development Goal
TRO	Test Requirement Optimisation
TRR	Test Redundancy Reduction
UN	United Nations

Chapter 1

Introduction

1.1 Problem statement

Today software testing is an inescapable part of any IT project. Testing may take place at several stages of a project, such as the development and pre-production stages. Some stages are even fully dedicated to testing, such as the user acceptance tests stage. The lack of testing can cause small to catastrophic software failures, and the costs of said failures are both economic and social. For example the economical cost of lack of software testing was estimated between $22.2 and $59.5 billion for the year 2002, for the United States of America alone [1]. A more recent review estimated the cost of poor quality testing in the United States of America for the year 2020 to $2.08 trillion [2]. With the ongoing trend of the digitalisation of the world systems, the potential costs of lack of testing only increase. Another example of the cost of lack of testing is the CrowdStrike Falcon failure on Microsoft Windows where validation tests let problematic data that caused the failure be sent on release, crashing affected computers [3]. Not only were companies affected by this failure, but also hospitals, highlighting the social costs of software failures. As such there is a need for a reliable, formal testing background to support the testing activities, at an affordable price for industries.

The project originates from the analysis of Prof. Karl Meinke that the field's literature lacks formality, impacting the scientific results.

1.2 Motivations

Testing is a vast scientific field that encompasses largely different paradigms, as may be observed from the numerous subfields of software testing: glass box testing, black-box testing, requirement testing and model-based testing to name but a few. In this work we focus on glass box testing, also called structural testing as it tests the structure of a given program. Curiously, glass box testing lacks formality, a fact that can have major impacts on the validity of the results of scientific papers. An example of the lack of formality is the key concept of coverage: when is an element of the program covered? We have not found a satisfying answer in the literature. As such one of the goals of this work is to establish an extensively formal approach for glass box testing.

Although needed for quality, test suites are not free to execute, they notably require hardware on which to run the tests, time and energy. As such many methods can be found in the literature to reduce the size of a test suite. Most of these methods attempt to solve the Test Redundancy Reduction (TRR) problem, where test cases that redundantly cover requirements w.r.t. the rest of the test suite are removed to form an optimised test suite. Another problem w.r.t. test suite size reduction is the Test Requirement Optimisation (TRO) problem, where the quality of the test requirements set is questioned. Some test requirements can be covered by all test cases of some other requirements, and may thus be safely removed from the test requirements set. Another goal of this work is to be able to formally reason on the relations between test requirements, attempting to solve the TRO problem.

Finally, writing test cases is a long and error-prone task. Automatic test case generation is thus a valuable feature and our last goal for this work.

1.3 Methods and concepts

This section presents the main concepts and methods that will be formally defined in Chapter 3.

The formality of our approach starts with the definition of the language for the programs we may test, basing ourselves on the book of Ehrig and Mahr [4]. We separate the syntax and semantics of the program's language. The syntactic aspect is defined by a first-order signature Σ containing the available sorts[*], and the constant, operation and relation symbols upon which we build terms and First-Order Logic (FOL) formulas. The semantic aspect is defined through a first-order Σ-structure M which has one domain per sort of Σ and associates a meaning to the symbols of Σ. Terms and formulas defined by the symbols of Σ can be evaluated into the domains and propositional constants of M respectively. Our model builds upon the ordered ring of integers with

[*] e.g. integers, floating-point numbers, etc.

integer division $\mathcal{Z}_{div}$, signature that can be extended to incorporate other sorts and symbols to suit the needs of the reader.

Software errors such as the typical division by 0, that may be encountered with the integer division of $\mathcal{Z}_{div}$, can be handled at several levels: the data-type level, the language's runtime level and the operating system level. Our approach handles errors at the data type level, where errors are evaluated to a certain value, possibly non-standard, in the domains of the chosen structure.

The glass box test requirements we will study are paths of a representative graph of a given program. Such representative graphs in our work are Control Flow Graphs (CFGs), directed graphs where nodes are labelled with either assignment statements or conditional statements, modelled in our work by a FOL formula. The control flows of the program are modelled by the edges of the graph. To mirror program execution, we define how to execute a CFG. In particular we are interested in the sequence of nodes that are executed during an execution of the graph for a given input state, called the execution path. A path, and by extension a test requirement, is covered if it is a subsequence of the execution path, and exited in a particular state. We also require for coverage that the next node to be executed is fixed, to define the control flow after the path is exited. Coverage is also defined with FOL formulas instead of states, and holds when all input states that satisfy the precondition formula cover the path and the postcondition formula is satisfied by the state at the exit of the path. The coverage relation with FOL formulas is denoted $G, v, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), \chi$ where G is the graph model of the program, v the node at which we start the execution, ϕ the precondition formula $p = (p_1, \ldots, p_n)$ the path that is covered, p_{n+1} is the next node to be executed, and χ is the postcondition formula.

Using the formal definition of coverage, we build a formal finitary proof system which can prove inference sequents of the form $G, v, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \chi$. Through the soundness of the calculus, it is known that $G, v, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \chi$ being provable by our calculus implies the analogous coverage truth $G, v, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), \chi$ holds. In some cases the calculus guarantees that ϕ is a weakest precondition formula for $(p_1, \ldots, p_n, p_{n+1})$, that is the formula is satisfied by only and all the input states that cover $(p_1, \ldots, p_n, p_{n+1})$ starting in v, exiting in a state satisfying postcondition formula χ.

The set of input states that cover a path, captured by the weakest precondition formula obtained by our calculus, let us infer relations between paths and by extension between glass box test requirements. A test requirement p strongly covers a test requirement q if the set of input states that cover p is a subset of the set of input states that cover q. Strong coverage implies that all test cases of p, which are input states restricted to the input variables of the graph, are also test cases of q and q can be safely removed from the test requirement sets. Another relation is weak coverage, where p weakly covers q if the intersection of the set of input states that cover p and q is not the empty set. Weak coverage implies that some test cases of p are also test cases of q, and one such common test case can be chosen to cover both requirements. The calculus can be automated to obtain test cases for a given set of test requirements.

1.4 Goals and hypothesis

In this section we refine the informal goals presented in Section 1.2 with the concepts presented in Section 1.3. The high-level goal of this project is to find a sound and finitary proof system to reason on glass box test requirement weakest precondition and redundancy, and to establish its properties. This has been divided into the following goals:

OB1. Establish a formal mathematical basis for glass box testing.

OB2. Find a sound finitary proof system for demonstrating redundancy between test case requirements as paths of a representative graph of a program and evaluate its properties.

OB3. Automate OB2's logic for test case construction for a specific path requirement.

We conjecture that the proof system of OB2 is incomplete in the general case, but may be complete when loops are limited to have no impact on the precondition formula obtained by the proof system. We hypothesise it is undecidable whether the conditions on loops mentioned above are met or not in the general case as there may be an infinite number of loops for which their impact on the precondition formula must be checked.

1.5 Delimitations

Other subfields of software testing, such as the commonly used in industry subfield of black-box testing, are not included in this project. Certain glass box coverage models including logic coverage models are not included in the project, as we define requirements as paths of a representative graph and not logical constraints. Logic coverage models include Modified Condition/Decision Coverage (MC/DC), a strict coverage required for critical systems in planes by the Federal Aviation Authority [5].

Our model of programs limits conditional statements to be side effect-free. The model is meant to capture software errors at the data type level. Although we present only the integer data type in this thesis, the model can be extended as defined in the methodology to incorporate data types such as floating-point numbers.

1.6 Structure of the thesis

Chapter 2 presents relevant background information about the mathematical foundations of testing, general testing background and links to the methodology of testing to the field of software verification. Chapter 3 presents the methodology of the project, including the mathematical definitions and theorems which constitute our main results. Chapter 4 presents the limitations of the project and addresses future work w.r.t. the project. Chapter 5 concludes the project and assesses the ethical and sustainable impacts of the work.

Chapter 2

Background and literature survey

This chapter provides an introduction to the mathematical concepts used in the methods and the general testing background, including works on the TRO and TRR problems. Finally, it presents links in methodology of testing to the field of software verification.

2.1 Mathematical concepts

The field of testing relies on a broad mathematical background, that is sometimes omitted in the testing literature. This section summarises the different mathematical concepts used in the thesis and their relevance to the problem.

2.1.1 Control flow graphs

A Control Flow Graph (CFG) is a graph representation of a program where the control flows can be deduced from the statements associated with the nodes, and the edges of the graph. In the field of glass box testing, test requirements are paths of a graph model of the tested program. CFGs are the *de facto* semi-standard graph representation of programs used in the literature and the industry.

CFGs were first defined by Frances E. Allen in [6] and have largely evolved since. Notably, we refine Ammann and Offut [7]'s approach* and

*We notably let the reader refer to Figures 7.16 to 7.24 for short examples of the authors' CFGs and Figures 7.12 and 7.13 for more complete examples.

make the CFG executable. We present in Fig. 2.1 an arbitrary CFG to detail our approach. Our CFGs allow for three types of nodes, nodes containing an assignment statement such as v_1, v_2, v_4, v_6, v_7 in Fig. 2.1, conditional nodes containing a First-Order Logic (FOL) formula such as v_0, v_3, v_5 and the termination pseudo-node τ, in orange in Fig. 2.1. The edges exiting a conditional node are labelled with $true$ or $false$, corresponding to whether the formula contained in the node evaluates to tt or ff at execution. The definitions that let us formally define a CFG by our approach are found in Chapter 3.

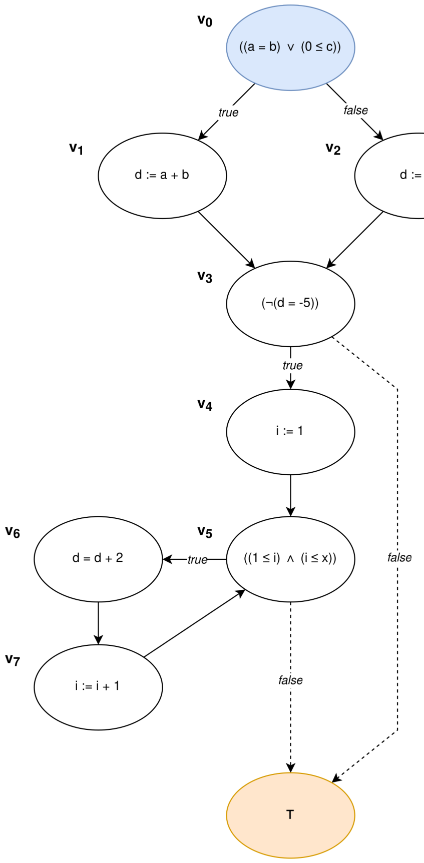


Figure 2.1: A simple and arbitrary CFG

We diverge from Ammann and Offut [7]'s approach in two major ways. First, the graphs of [7] allow no statement to several statements per node, which makes it unclear which statement(s) caused an error in a node; as such we require a single statement per node. Secondly, the graphs of [7] have guards on the edges, which may be boolean guards, control flow instructions such as ”break” and ”continue” and Java exceptions*, but with this approach it is possible to have non-determinism in the control flows, in the case these three type of guards can be found for the exits of a single node. Instead, we label the edges with $true$ or $false$ as described above, ensuring determinism in the control flows.

2.1.2 Structures and data types

Data types and abstract data types have their foundation in the concept of mathematical structures. This section will focus on the informal presentation and motivation of the usage of these concepts, and we will formally introduce concepts when not defined in the methodology. We base ourselves on the book of Ehrig and Mahr [4].

As usual in computer science we should clearly separate syntax and semantics. In this work we introduce signatures as syntactic elements and structures as semantic elements. A first-order signature is a pair (S, Σ) where S is called a set of sorts and Σ is a set of constant, operation and relation symbols (respectively denoted by $\Sigma_{\lambda,s}$, $\Sigma_{w,s}$ and Σ_w). Sorts in S are domains, e.g. integers, floating-point numbers, etc. A constant symbol in $\Sigma_{\lambda,s}$ has no argument and is of sort $s \in S$. An operation symbol in $\Sigma_{w,s}$ has argument sorts $w = s_1 \ldots s_n \in S^+$ and is of sort $s \in S$. A relation symbol in Σ_w has argument sorts $w = s_1 \ldots s_n \in S^*$. For example addition and the order relation ”less than” may be introduced respectively in $\Sigma_{int\ int,int}$ and $\Sigma_{int\ int}$. For constants, the notation is $\Sigma_{\lambda,s}$ and for propositional constants (notably $true$ and $false$) the notation is Σ_λ. By abuse of notation we will denote Σ a signature in the remaining of the section. As mentioned prior, signatures are syntactic elements, they can be compared to interfaces of the Java programming language as they define what is possible to do, but not what they do.

Structures are semantic elements built round a signature, they provide the semantics associated to the syntax provided by the signature. For each sort of a first-order signature Σ, a first-order Σ-structure has carriers, concrete domains with for example $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ for the int sort. We note c_M the interpretation of the constant c in the structure M. Similarly, all operation symbols have their interpretation in the structure. While signatures are similar to interfaces, structures are close to concrete classes in Java.

Terms are constructs of constant symbols, operation symbols and some

*The graphs of Figures 7.12, 7.13, and 7.16 to 7.24 are for the Java programming language.

variables of adequate sorts with the operation symbols. The set of terms $T(\Sigma, X)_s$ of sort $s \in S$ for some S-sorted signature Σ and some S-sorted set of variables $X = \{X_s | s \in S\}$ is formed of constants, variables, and operations of sort s for some adequate sub-terms.

Terms of a structure M can be evaluated to obtain a certain value in the carriers of M. A family of evaluation functions for a variable mapping $\alpha : X \to M$, denoted $eval_{M,\alpha} : T(\Sigma, X) \to M$ associates a term with its interpretation in the carriers of M. The process is recursive, if a term has sub-terms then the evaluation replaces the sub-terms by their interpretations. The variables are replaced by their associated value in the variable mapping α. Evaluation is necessary for us to learn the value to assign to a variable for instance.

First-order formulas are constructs of relation symbols applied to well-formed terms, logical connectives and quantifiers. Atomic formulas are a subset of FOL formulas formed only by the relation symbols in Σ applied to well-formed terms. Then for any formulas ϕ, χ the conjunction and disjunction of ϕ and χ are FOL formulas. Similarly, the negation or quantification of a formula is itself a formula. We denote $FOL(\Sigma, X)$ the set of FOL formulas. The evaluation function for terms can be extended in a similar manner to evaluate formulas. Formulas will be used to model conditional statements.

Only a fraction of the possible structures are of interest to testers. A notable case of valid but uninteresting structure is one that maps the propositional constant $true$ to ff, that to our knowledge no modern language would use. Restrictions to obtain interesting structures and a method to expand a signature with additional sorts and symbols will be presented in the methodology.

Given a signature Σ, a Σ-structure M is a concrete data type of Σ. However, there is an infinite number of Σ-structures M. As such we introduce a set of equations E where equations are of the form $L = R$ with L and R two terms of the same sort. The equations of E are valid in M iff for all evaluation functions the evaluation of L is equal to the evaluation of R; it is also said that M satisfies E. An equational data type specification is a pair (Σ, E); it is possible to obtain the set of Σ-structures for which the E is satisfied through a factory. This set is the loose semantics of the equational data type specification (Σ, E), and no structure outside of this set is relevant for a tester. The reader may select one structure in the loose semantics of (Σ, E) as a structure for the graphs of our approach.

2.1.3 Proof systems

Considering whether logical statements about a structure (e.g. $PA = (\mathbb{N}, 0, +1, +, *)$) are correct or not is a problem of logic, that lets us find interesting facts about a system. A proof system is an inference machine that can decide such property. We base ourselves on the chapter "General Proof Systems: Syntax and Semantics" by Wasilewska [8]*. Formally, a proof system is a tuple

$$S = (\mathcal{L}, \mathcal{E}, LA, \mathcal{R})$$

where $\mathcal{L} = (\mathcal{A}, \mathcal{F})$ is a formal language with an alphabet $\mathcal{A}$ and a set $\mathcal{F}$ of well-formed formulas (e.g. of zeroth-order logic, FOL, etc.), $\mathcal{E}$ is a set of expressions of S, $LA \subseteq \mathcal{E}$ is a non-empty set of logical axioms of the system, and $\mathcal{R}$ is a set of rules of inference. Expressions in $\mathcal{E}$ are built from formulas in $\mathcal{F}$ and additional symbols that are not part of the language $\mathcal{L}$, extending the language to suit the needs of the proof system. For example, expressions can be sequences of formulas, which are notably used by Gentzen's LK and LJ sequent calculi [9, 10]. When there is no need for extensions of the language, proof systems use $\mathcal{E} = \mathcal{F}$. The set of logical axioms is assumed to be finite. The rules of inference in $\mathcal{R}$ are comprised of finite premises (the system is finitary), with at least one premise per rule, and a conclusion. A rule $r \in \mathcal{R}$ of premises $P_1, \ldots, P_m$ and conclusion C is denoted as

$$\text{r}\frac{P_1 \qquad \ldots \qquad P_m}{C}$$

When all the premises $P_1, \ldots, P_m$ are provable then the conclusion C is provable itself. C is called a direct consequence of $P_1, \ldots, P_m$ by virtue of r.

A formal proof in a proof system $S = (\mathcal{L}, \mathcal{E}, LA, \mathcal{R})$ is a sequence of expressions $E_1, \ldots, E_n$ with $n \geq 1$ the length of the proof, such that $E_1 \in LA$ and for each $1 < i \leq n$ either $E_i \in LA$ (axioms are always provable) or E_i is a direct consequence of some expressions $E_j, 1 \leq j < i$ by virtue of a rule of inference $r \in \mathcal{R}$. We use $\vdash_S E_n$ to denote that E_n has a formal proof in S.

Three more properties of proof systems are of note, soundness, completeness and decidability. To formally define them, we introduce M some semantics which defines what is true for our system and notably impacts what axioms we will choose, T_M the set of true formulas in M, and P_S the set of provable expressions in S. Then, we define the satisfiability relation $\models_M E$ with $E \in \mathcal{E}$ an expression, indicating that E is true in M. A proof system $S = (\mathcal{L}, \mathcal{E}, LA, \mathcal{R})$ is sound under semantics M if for any expression $E \in \mathcal{E}$

*Note this is the corrected version of the chapter, the original one can be found here.

then $\vdash_S E \implies \vDash_M E$. This implies that the inference rules are correct, and $P_S \subseteq T_M$. A proof system is complete if $P_S = T_M$, i.e. $P_S \subseteq T_M$ and $P_S \supseteq T_M$. As such there must be $\vDash_M E \Leftrightarrow \vdash_S E$. $P_S \subseteq T_M$ is given for a sound proof system, $P_S \supseteq T_M$ is notably harder to prove. Henkin's proof style [11] is one method to prove completeness. A proof system is decidable if there is a procedure that can decide whether there is a proof or there is no proof for any expression $E \in \mathcal{E}$. If the proof system is decidable it can be fully automated.

Many interesting formal theories (proof systems with specific axioms valid in the universe we wish to describe) are incomplete, as shown by Gödel's first incompleteness theorem [12]. It implies that theories that can prove some elementary theorems of arithmetic are incomplete. Since our signature for the programming statements contained in our graphs incorporates the sort of integers, it is likely that our model will be incomplete for any signature extending the minimal one we define.

2.2 Testing

2.2.1 Test coverage

Testing is such a vast field it can be quite complicated to decide which sub-field (glass box testing, black-box testing, requirement testing, model-based testing, etc.) to choose to test a program. Even when restricting to glass box testing, when should a tester stop to add test cases and how should they estimate the quality of the test suite? As such an indicator of the quality of the tests and an indicator to know when to stop testing is used: coverage. Coverage models (also called coverage criteria by Ammann and Offut [7]) are rules or collections of rules that define a set of test requirements to satisfy. In this section we will use the definition of a test requirement by Ammann and Offut [7], "A test requirement is a specific element of a software artifact that a test case must satisfy or cover."*. In the context of glass box (or structural) testing, elements of the graph corresponding to the program under test are the test requirements. We do not address black-box testing, which concerns test constructed without reference to the internal code structure of the program.

Coverage models can be divided into categories, notably control flow, logic and data flow coverage models. Control flow coverage models require that certain paths in the graph of the program are satisfied. Some examples

*We formally define glass box test requirements in Definition 3.25.

include Node Coverage (NC) where requirements are the set of paths containing a single node and Edge Coverage (EC) where the edges of the graph are the requirements. Logic coverage models exercise conditional branches in a certain way. Predicate Coverage (PC) (also called Branch Coverage) requires to reach all conditional statements, and for each statement a test case must have it evaluate to false, and one to true. More elaborate logic coverage models include MC/DC which is required for safety-critical plane software by the Federal Aviation Authority [5]. Data flow coverage models require to satisfy certain paths based on whether they write to or read some variables from memory. For instance, All-uses Coverage requires one path that writes to a variable in their first node and reads it in their last node without writing to the variable in between to be satisfied, for some variable, first node and last node triples. All-uses Coverage can be used by compilers to check for unused variables in the program. The previously mentioned coverage models are general, but more domain-specific models exist, such as the coverage models for automatic security of web applications presented in Dao et al. [13], the coverage models for graphical user interface testing by Memon et al. [14], or the adapted control flow and state coverage models of Devroey et al. [15].

Quite often the coverage is not satisfied, as shown in the survey of Prause et al. [16]. Even advanced projects, in the field of space software projects, do not fulfil the coverage model. As such the notion of coverage level is introduced by Ammann and Offut [7] as the ratio of the number of test requirements satisfied by a test suite over the size of the test requirements set. Full coverage is not often achieved, partly due to the cost of implementing the remaining test cases to satisfy full coverage compared to their apparent capacity to reveal bugs, but also due to infeasible requirements. The latter part prevents any test suite to reach full coverage.

The informal model of Ammann and Offut [7] on which we base ourselves and our model have differences due to the formality of our approach. The first difference is how we build our CFGs. We discuss these differences in Section 2.1.1. Secondly, Ammann and Offut [7] considers a test requirements set to be satisfied by a test suite iff at least one complete execution of a test case (referred to in [7] as a test path) contains the test requirement; this process is referred to as touring*. Whether a path is satisfied at the entry of the last node or at the exit of the last node is thus unclear. Our model considers a path satisfied after the last node of the path is exited. The authors also introduce the concept of touring with sidetrips, when every edge of the test requirement is contained in the test path in the same order as in the test requirement. Touring

*We refer to a similar concept as path matching.

with sidetrips is meant to overcome certain dead requirements, for example when a loop body is guaranteed to be executed once before exiting the loop (in opposition to not satisfying the loop guard in the first place); however we consider this does not fit the goals of our work and we keep the strict notion of matching. Our work can identify dead test requirements when the weakest precondition formula obtained for the requirement implies $false$.

2.2.2 Redundancy reduction

Test case construction, whether manual or automatic, leads to redundancy in the test suite. Redundancy of test cases occurs when at least 2 test cases satisfy a common test requirement. Test case execution costs resources such as computation time and memory, and as such the industry tries to reduce the number of test cases in a test suite.

Ammann and Offut [7] defines a minimal test suite TS as one satisfying all requirements of a set of test requirements TR such that removing any test case in TS yields a test suite that does not satisfy all requirements of TR. Furthermore, the authors define a minimum test suite TS satisfying a given test requirements set TR as a set of test cases that satisfies all test requirements of TR and so that all other test suites satisfying TR are of greater or equal cardinality. Finding a minimal test suite is achievable with relative ease, whereas finding a minimum test suite is much harder and most methods only approximate a minimal test suite.

The literature contains two different approaches to reduce the size of the test suite: Test Requirement Optimisation (TRO) and Test Redundancy Reduction (TRR).

To our knowledge, few articles question the TRO problem. This is supported by Chen et al.'s claim that "The test suites are always reduced by analysing the satisfiability relation between testing requirements and test cases." [17]. Ammann and Offut [7] presents the simple approach of coverage subsumption to redundancy. A coverage criterion C_1 subsumes coverage criterion C_2 iff every set of test cases satisfying C_1 also satisfies C_2. It is noted to be a very rough criterion by the authors. We leave to the reader to prove that EC subsumes NC. Chen et al. [17] notes that "In some cases, the relationship of testing requirements can be captured before the test case generation by requirement engineering, semantic analysis, program analysis, domain knowledge, testing history and etc." and further proposes a method for requirement optimisation by contraction. The authors introduce a bigraph (T, R, E) where $T = \{t_1, \ldots, t_n\}$ is the set of test cases and $R = \{r_1, \ldots, r_m\}$

the set of test requirements where each r_i is a subset of T. The authors propose two possible relations on requirements: given $r, r' \in \mathcal{P}(T)$, r and r' are intersectant, noted $r \bowtie r'$ iff $r \cap r' \neq \varnothing$, and r is subsumed by r', noted $r \preceq r'$ iff $r \subseteq r'$. A test requirement $r \in R$ is said to be 1-1 redundant in R if $r' \preceq r$ for some $r' \in R$. The authors prove that removing the 1-1 redundant test cases does not change the satisfiability of the resulting test suite for the test requirements set, and the same property is proved by replacing intersectant test cases $r, r' \in R$ by $r \cap r'$. Our work in this thesis can determine the relations between requirements for structural testing.

Most approaches in the literature attempt to solve the TRR problem. The literature on TRR goes as far back as the 1990s and presents the problem as follows from Harrold et al. [18]

Given: A test suite TS, a set of requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$ one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all of the r_i's.

Finding the optimal solution to this problem is NP-complete [19]. Various approximations can be found in the literature and the methods to obtain them can be divided in four categories: exact methods, heuristic-based methods, metaheuristic-based methods and Artificial Intelligence (AI)-based methods. We will thereafter present two articles on the TRR problem and list some other papers without analysing them in-depth.

The heuristic method by Harrold et al. [18] approximates the optimal TRR solution by selecting representative test cases to cover the requirements. It only applies to structural coverage models. This method first considers the T_i's of cardinality one, i.e. containing a single test case, and adds each test case to the set of representative test cases. All T_i's containing the selected test cases are marked. The method then proceeds to the unmarked T_i's of higher cardinality as follows: the test case t_i appearing in the largest number of unmarked T_i's of current cardinality c is selected. If there is a tie, the algorithm recursively examines the unmarked T_i's of cardinality $c' > c$ until one test case in the tie appears in strictly more T_i's than the others, and adds it to the representative set. Then all T_i's containing t_i are marked and the algorithm repeats the process at cardinality c until all T_i's of cardinality c are marked. The previous steps are repeated with incremented cardinality, until all T_i's are marked. The representative set of test cases is the minimised test suite. The runtime complexity of this algorithm is $\mathcal{O}(n(n+nt)C)$ where C is the maximal cardinality of the T_i's, n the number of T_i's and nt the number of test cases t_i. The method can be used when considering several coverage models by computing the representative set for one coverage, and use it as the basis of the second representative set for a new coverage, and so on. This method is related to our concept of weak coverage as test cases are considered for their capacity to satisfy several test requirements.

Fraser and Wotawa [20] base their exact approach to approximate the optimal solution of the TRR problem on model checkers and Kripke structures. A model checker takes as input a finite-state representation of a program and a temporal logic property. The model checker will verify on the entire space if the property is satisfied, if not it returns a sequence of states leading to the invalid state not satisfying the property. Such a sequence is called a counter-example or a trace. The authors' method considers the test cases to be traces generated by trap properties [21]. The behaviour of the system under test can be represented by a Kripke structure. Formally, a Kripke structure is a tuple $K = (S, s_0, T, L)$ with S the set of states, $s_0 \in S$ the initial state, $T \subseteq S \times S$ the transition relation and $L : S \to 2^{AP}$ the labelling function mapping each state to a set of atomic propositions holding in this state, where AP is the countable set of atomic propositions. Paths in a Kripke structure K are finite or infinite sequences of states $\{s_0, s_1, \ldots\}$ such that $\forall i > 0 : (s_i, s_{i+1}) \in T$. Test cases are defined as finite prefixes of a path p of a Kripke structure K. Redundancy of test cases is defined as the existence of a common prefix of at least two test cases of same initial state. As test case removal lowers the test suite fault detection capability [22, 23, 24], their approach does not only remove test cases but modifies them as follows: for each test case t of length n (the number of transitions in t) in test suite TS, search for other test cases $t_2 \in TS$ such that t_2 has a common prefix with t of length n, and repeat for $n-1$, down to 1 until a prefix of length $1 \leq n' \leq n$ is found. If such a prefix of length n' exists, and there exists $t_3 \in TS$ such that t_3's last state is equal to the last state of the prefix s_p, the postfix of t_1 is appended to t_3. Otherwise, a non-deterministically chosen $t_4 \in TS$ is used as the basis to generate a path from t_4's end to s_p with the model checker, called a glue sequence. Then t_4 is appended the glue sequence and then the postfix.

Other methods to approximate the optimal solution of the TRR problem include the exact approach by Chen, Zhang and Xu [25] that relaxes a 0-1 Integer Linear Programming (ILP) algorithm not suited for efficient computations for large test suites to a Degraded Integer Linear Programming

(DILP) algorithm. Given a set of test cases and a boolean matrix of the satisfaction of requirements by test cases the DILP algorithm gives a vector solution of values in $[0;1]$ where each index corresponds to a test case. Then it computes a score for each test case, and set the test case with the highest score to 1 in the output vector. The process is repeated until the output vector contains only 0 or 1 as values, thus being a solution for a 0-1 ILP algorithm. The test cases associated with 1 in the final output vector form the optimised test suite. Bajaj and Sangwan [26] propose a Gravitational Search Algorithm (GSA) inspired from the gravitational laws of physics, classified as a metaheuristic algorithm. The GSA algorithm orders a set of test cases based on their potential to reveal faults. The algorithm associates a mass to each test case, in turn applying a force of attraction to the other test cases. After a set number of iterations, the algorithm outputs as test suite the first set of test cases of highest potential to satisfy the coverage model. The authors update the gravitational constant and thus the force of attraction at each iteration to avoid local minima. Their first approach updates the constant with a random-generated value, and the second with a chaotic map. Hooda and Chhillar [27] introduce an AI method using an Artificial Neural Network (ANN) to reduce redundancy in a test suite. The authors parse UML 2.0 activity and statechart diagrams to generate test cases. This initial test suite is used as input for a Genetic Algorithm (GA) that generates new test cases. A trained back-propagation ANN decides whether the GA-generated test cases are redundant with the initial test suite, and the non-redundant test cases are added to the initial test suite.

2.2.3 Weakest preconditions

Whereas the previously mentioned works describe how to choose and satisfy test requirements, there remains the question of what output state may be acceptable after the execution of the test, also called the oracle problem. Dijkstra [28] introduced the concept of predicate transformers, logics that given a program S and a postcondition R obtain a precondition P so that executing S with any input state satisfying P yields an output state satisfying R. P is the weakest precondition for S and R iff the input states p satisfying P are the ones that guarantee S will terminate and the output state r satisfies R for any p. Dijkstra proposes a set of axioms and rules for a weakest precondition calculus, proved correct by Hoare in [29]. Later in the methodology, we will compare our calculus to a weakest precondition calculus by Basu and Yeh [30] inspired by Dijkstra's axioms.

The weakest preconditions in our work differ from that of Dijkstra's in [28] and Basu and Yeh's in [30] in that they concern paths of a graph model of a program. The weakest precondition to cover a path p when executing the graph, noted WP_{p,v_0} is the set of input states that execute p. Obtaining WP_{p,v_0} let us infer the test cases for p. Our calculus, defined in Section 3.5, can obtain weakest precondition formulas, FOL formulas that are satisfied by all and only the states of WP_{p,v_0}. It may be noted that a path of our approach fixes part of the control flows that may be present in an execution of the graph that covers the path*. Instead in [28, 30] all terminating executions are considered, such as if we considered all paths from the entry node of the graph to the termination pseudo-node. Furthermore, our calculus is designed for test case construction rather than to solve the oracle problem and as such we allow any postcondition. We discuss in the future work the challenges we identified for a postcondition calculus that may solve the oracle problem.

In the case of this work, weakest preconditions let us infer relations between test requirements. If the weakest precondition to execute a path p, WP_{p,v_0}, is a subset of the weakest precondition to execute a path q, WP_{q,v_0}, then all executions that cover p also cover q. As such q can be safely removed from the test requirements set. Our definition of weakest preconditions does not require termination. However it is possible to include the termination pseudo-node τ as the last node of a path to require termination. In this case, the halting problem becomes relevant, as discussed in Chapter 4.

2.3 Methodology and software verification

While verification is powerful, "proofs cannot substitute tests" [31] as verification is usually not performed with every component of the program's environment that can impact its behaviour, such has the operating system, compiler, hardware, etc. In contrast to verification, testing encompasses the stack of the program. Yet several concepts and tools used in the field of software program verification are also used for testing.

One such example is *KeYTestGen*, a program in the KeY project whose associated book chapter by Ahrendt et al. [31] presents it. *KeYTestGen* is an automatic test case generation software that is embedded in a program logic for formal verification, a superset of FOL. First the program verifies the Method Under Test (MUT), if verification failed then the program generates

*In other words, a given path implies a sequence of branches taken during any execution for the path.

a small-size test suite with counter-examples* to help locate the problems in conjunction with a debugger. If instead verification succeeded, the program generates a large test suite with high coverage to further increase confidence in the MUT. Test case generation by *KeYTestGen* starts by constructing a proof tree where branches are symbolic executions of the program. Through construction, conditions on the initial state of the program to reach a certain branch are accumulated, which allow to generate restrictions for test cases. By selecting one mapping of input variables to values that satisfy the above constraints, a concrete test case is generated. *KeyTestGen* permits to generate glass box test suites for logic coverage models, whereas this work focusses on control flow and data flow coverage models. The authors point to the assessment of Cadar et al. [32] for a survey of symbolic execution-based test case generators.

Other uses of common verification tools for software testing include the adoption of symbolic evaluation and theorem provers to detect infeasible paths, as detailed in the work of Goldberg at al. [33]. The system under test is parsed into an abstract syntax tree which will be given as input to the theorem prover. With a set of paths r from a starting node s to an ending node e and a formula φ, the theorem prover verifies if there exists a path in r such that formula φ holds when e is reached. This is done by creating a FOL formula through parsing the graph with symbolic evaluation. The possible answers are that there exists such a path, that there is not, or that the search is inconclusive. At the time of the work, such use was unpractical due to the length of the formulas given to the theorem prover. The authors used simplification rules to keep the formula manageable in acceptable time by the theorem prover. In their work, the authors mention the problem of loops that modify variables in φ and cannot be finitely inlined efficiently, as they could represent infinitely many paths. In such case they consider the loop modifies the variables in an arbitrary way. A similar problem is reported for complex functions. In the future work section, the authors propose the use of high-order functions to resolve this problem.

Gladisch [34] proposes a different approach to the complex loops and functions problem. The author's method computes a precondition before a loop or method is executed so that a certain path is executed after the execution of the loop or method. The method is similar to the weakest precondition computation introduced by Dijkstra [28]. The Full Disjunctive Branch Precondition (F-DBPC) for a program p and a branch condition φ to execute a certain branch in p is the conjunction of $pre \rightarrow \langle p \rangle post$ (similar to the Hoare triple $\{pre\}p\{post\}$ but also asserts that p terminates), and the

*See Section 2.2.2's description of Fraser and Wotawa's work for details.

DBPC $pre \to \{M := M_{sk}\}(post \to \varphi)$. In the DBPC function symbols in M that can be modified by p are replaced by new symbols from M_{sk} in $post$ and φ. F-DBPC is useful when instead of a branch condition φ its negation $\neg\varphi$ is used in the F-DBPC. In this case, if the F-DBPC is validated by a theorem prover, then either the precondition or the path condition is unsatisfiable, so the path is infeasible under the program's contract.

2.4 Summary

Programs are constructs of structures and control flows, which in turn can be used to model them as executable graphs. From this mathematical basis, proof systems let us infer facts as expressions of the formal language we created with control flows and structures. We aspire to check the program's behaviour on some inputs, and employ coverage models to obtain testing requirements. Test suites for a given coverage are often not optimised. Two main approaches to this problem are the optimisation of the test requirements such as in the work of Chen et al. [17] and the optimisation of the test suite based on test cases satisfying multiple test requirements. The latter approach can be solved with multiple methods, be they exact, heuristic, metaheuristic or AI-based. Weakest precondition calculi let us obtain the set of inputs that satisfy a given postcondition for a given program, which in turn solves the oracle problem. Our work proposes an approach to study the weakest preconditions as to infer relations between structural testing requirements. Some concepts used in the field of testing are taken from the field of software program verification, such as the use of proof trees and symbolic evaluation for test case generation and infeasible paths detection.

Chapter 3

Methodology

This chapter provides an overview of the research methods used in this thesis. Section 3.1 introduces the concept of software errors. Section 3.2 presents the signatures and structures. Section 3.3 presents the CFGs. Section 3.4 presents the test requirements. Section 3.5 introduces the inference rules of our calculus. Section 3.6 determines the weakest preconditions obtained by our calculus. Finally, Section 3.7 compares our calculus to the literature.

The definitions, propositions, lemmas, theorems and their proofs constitute the results of this work. The chapter presents examples and graphs to simplify the understanding of the results for the reader.

3.1 Software errors

There are as many definitions of software errors as persons using the term.

Definition 3.1. We define a software error on a program as an incorrect behaviour of the program according to the program specifications of the person interacting with the program. ■

The specifications of Definition 3.1 depend on the person interacting with the program, e.g. developer, user, client. Program specifications may be documented or not documented, precise or informal. In the case of informal non-documented specifications, software errors are unclear and the task of testing is convoluted. Some categorisations of software errors with regard to the program specifications include:

- Incorrect termination behaviour (the program terminates when it shouldn't, or does not terminate when it should)

- Operational errors, including the execution of incorrect expressions, such as x/y when $y = 0$ and both x and y are integers, leading to incoherent states*
- Bound errors, such as integer overflow
- Requirement errors, that suppose the program terminated in a coherent state, but the results or side effects do not correspond to the specifications or oracle
- Environmental errors, when the environment of the program causes an invalid behaviour, such as forced termination by the operating system, or an external input that is not as expected

3.2 Signatures and structures

Data types form a founding concept of programs, they set syntax and semantics on variables, defining what they are and what can be done with them, with dedicated symbols and names. A formal definition of data types involves signatures as their syntax, and structures as their semantics.

3.2.1 Languages foundations

To establish data types, we must first define a language to build upon.

A signature syntactically defines a language, by containing the available types, certain values' symbols such as 0 and operation and relation symbols such as + and $<$.

Definition 3.2 (S-sorted first-order signature)**.** An S*-sorted first-order signature* is a pair (S, Σ) where:

(i) S is a non-empty set of sorts, $S = \{s_1, \ldots, s_n\}$

(ii) Σ is a set of constant, operation and relation symbols, formed by the union of pairwise disjoint subsets:

(a) $\Sigma_{\lambda,s}$, the set of constant symbols of sort $s \in S$

(b) $\Sigma_{w,s}$, the set of operation symbols of sort $s \in S$ with argument sorts $w = s_1 \ldots s_n \in S^+$

*Certain programming languages prefer to crash when such an error is reached, but others consider a value such as not-a-number (NaN), or $Infinity$.

(c) Σ_w, the set of relation symbols of argument sorts $w = s_1 \ldots s_n \in S^*$, in particular if $w \in S^0$ the symbol is a propositional constant symbol and we denote Σ_λ the set of propositional constant symbols

■

It is immediate that $\Sigma_\lambda \subseteq \Sigma_w$. By abuse of notation, in the sequel we let Σ an S-sorted signature.

Structures are the semantics of the language defined syntactically by a signature. In terms of usual programming languages, a structure is the implementation of a language. For example most languages associate the value 0 for integers to the constant symbol 0 of the same type.

Definition 3.3 (Σ-structure)**.** A *many-sorted first-order* Σ*-structure* M is a triple $(S_M, \Sigma^M_{w,s}, \Sigma^M_w)$ where:

(i) S_M is an S-indexed family $\langle M_s | s \in S\rangle$ of non-empty sets, where for each $s \in S$, M_s is called the *carrier* or *domain* of M of sort s

(ii) For any $w = s_1 \ldots s_n \in S^*$, $s \in S$, $\Sigma^M_{w,s}$ is an $S^* \times S$-indexed family of sets of constants and sets of functions. For each sort $s \in S$

$$\Sigma^M_{\lambda,s} = \{c_M | c \in \Sigma_{\lambda,s}\}$$

where $c_M \in M_s$ is a *constant* of sort $s \in S$ which interprets the constant symbol $c \in \Sigma_{\lambda,s}$ in the structure M. For each $w \in S^+$ and each sort $s \in S$

$$\Sigma^M_{w,s} = \{f_M | f \in \Sigma_{w,s}\}$$

where $f_M : M^w \rightarrow M_s$ is termed an *operation* or *function* with domain

$$M^w = M_{s_1} \times \ldots \times M_{s_n}$$

codomain M_s and of arity n which interprets the function symbol f in the structure M.

(iii) For any $w = s_1 \ldots s_n \in S^*$, $n \geq 0$, Σ^M_w is a S^*-indexed family of propositional constants and predicates. For each $w \in S^*$

$$\Sigma^M_w = \{r_M | r \in \Sigma_w\}$$

where $r_M \subseteq M^w = M_{s_1} \times \ldots \times M_{s_n}$ is termed a *predicate* of arity n which interprets the relation symbol r in the structure M. In particular we term predicates of arity 0 *propositional constants*.

We denote $Mod(\Sigma)$ the set of all many-sorted first-order Σ-structures. ■

3.2.2 Assignments

In addition to signatures and structures, usual programming languages contain variables, and provide a form of memory accessible through the variables. We hereafter define assignments as the ”memory” of the program for a defined language, mapping variables to values.

Definition 3.4 (Variables and assignments)**.** Let $X = \langle X_s | s \in S \rangle$ be an S-indexed family of sets $X_s = \{x_1, x_2, \ldots\}$ of variable symbols of sort s. Let $M \in Mod(\Sigma)$ be a many-sorted first-order Σ-structure. An *assignment* or *variable binding*[*] is an S-indexed family

$$\alpha = \langle \alpha_s : X_s \to M_s | s \in S \rangle$$

We let $[X \to M]$ denote the set of all assignments. ■

The family of variable symbols can contain more than the variables of a program, for example the family can contain a symbol $error$ of the integer sort int which may be assigned 0 if no error has occurred, and 1 if an error did occur (such as a division by 0).

Substitution in an assignment is a mechanism to update the assignment, which will be required by our assignment statements in the next section.

Definition 3.5 (Substitution in an assignment)**.** Let $M \in Mod(\Sigma)$ be any many-sorted first-order Σ-structure. Let $a \in M_s$ a value of the carrier of M of sort $s \in S$. Let $X = \langle X_s | s \in S \rangle$ be a family of sets of variable symbols and let $x \in X_s$ a variable symbol of sort s. Let $[X \to M]$ the set of all assignments w.r.t. M and X and let $\alpha = \langle \alpha_s : X_s \to M_s | s \in S \rangle \in [X \to M]$ be an assignment. The *substitution* of the mapping of variable x in assignment α by the value a, denoted $\alpha[x/a]$ is the assignment $\alpha' \in [X \to M]$ such that

(i) for all $s' \in S$ and all $y \in X_{s'}$ such that $y \neq x$ or $s' \neq s$, $\alpha'_{s'}(y) = \alpha_{s'}(y)$

(ii) $\alpha'_s(x) = a$

■

[*] Later in the context of CFGs we may refer to assignments as *states*.

3.2.3 Terms, formulas and evaluations

Signatures and structures define operations, but they yet lack how to define syntactically valid expressions.

Terms are the first valid expressions we will define. They are combinations of constant symbols (such as the 0 symbol) and operation symbols. Terms are of a certain sort.

Definition 3.6 (Terms)**.** Let $X = \langle X_s | s \in S \rangle$ be an S-indexed family of sets $X_s = \{x_1, x_2, \ldots\}$ of variable symbols of sort s. The set $T(\Sigma, X)_s$ of all terms over Σ and X of sort s for any $s \in S$ is defined inductively as follows

(i) if $c \in \Sigma_{\lambda,s}$ then $c \in T(\Sigma, X)_s$

(ii) if $x \in X_s$ then $x \in T(\Sigma, X)_s$

(iii) if $t_i \in T(\Sigma, X)_{s_i}$ for $i = 1, \ldots, n$ and $f \in \Sigma_{s_1 \ldots s_n, s}$ then

$$f(t_1, \ldots, t_n) \in T(\Sigma, X)_s$$

(iv) there are no other terms in $T(\Sigma, X)_s$

■

Terms are syntactical. Evaluating them in a structure and with an assignment lets us obtain a value in the corresponding carrier of the structure.

Definition 3.7 (Evaluation of terms)**.** Let $M \in Mod(\Sigma)$ be any many-sorted first-order Σ-structure. Let $X = \langle X_s | s \in S \rangle$ be a family of sets of variable symbols. Let $\alpha = \langle \alpha_s : X_s \to M_s | s \in S \rangle$ be an assignment. The *evaluation function of terms* $eval^s_{M,\alpha} : T(\Sigma, X)_s \to M_s$ is defined inductively as follows

(i) $eval^s_{M,\alpha}(c) = c_M$ for any $c \in \Sigma_{\lambda,s}$

(ii) $eval^s_{M,\alpha}(x) = \alpha(x)$ for any $x \in X_s$

(iii) $eval^s_{M,\alpha}(f(t_1, \ldots, t_n)) = f_M(eval^{s_1}_{M,\alpha}(t_1), \ldots, eval^{s_n}_{M,\alpha}(t_n))$ for all $f \in \Sigma_{w,s}$, all non-empty words $w = s_1 \ldots s_n$, all terms $t_i \in T(\Sigma, X)_{s_i}$ and $1 \leq i \leq n$

The family of evaluation mappings for terms w.r.t. M and α is $eval_{M,\alpha} = \langle eval^s_{M,\alpha} | s \in S \rangle$. ■

Evaluation is side effect-free.

Other valid expressions are First-Order Logic (FOL) formulas, differing from terms as they contain relation symbols.

Definition 3.8 (First-order logic formulas)**.** Let $X = \langle X_s | s \in S \rangle$ be a family of sets of variable symbols. The *set of First-Order Logic (FOL) formulas over* Σ *and* X, denoted $FOL(\Sigma, X)$ is defined inductively as follows

(i) if $r(t_1, \ldots, t_n) \in \Sigma_w$ for $w \in S^*$ and $t_i \in T(\Sigma, X)_{s_i}$ for $i = 1, \ldots, n$, then $r(t_1, \ldots, t_n) \in FOL(\Sigma, X)$

(ii) if $\phi \in FOL(\Sigma, X)$ then $(\neg \phi) \in FOL(\Sigma, X)$

(iii) if $\phi, \chi \in FOL(\Sigma, X)$, then $(\phi \wedge \chi) \in FOL(\Sigma, X)$ and $(\phi \vee \chi) \in FOL(\Sigma, X)$

(iv) if $r(t_1, \ldots, t_n) \in FOL(\Sigma, X)$ then for any $x \in X$ then $(\exists x. \, r(t_1, \ldots, t_n)) \in FOL(\Sigma, X)$ and $(\forall x. \, r(t_1, \ldots, t_n)) \in FOL(\Sigma, X)$

The formulas of (i) are termed the *atomic formulas*. The formulas closed under (i), (ii) and (iii) are termed the *quantifier-free formulas*. We denote $QFFOL(\Sigma, X)$ the set of quantifier-free formulas.

All occurrences of all variables of any quantifier-free formula are said to be *free*. In a FOL formula of the form $(\exists x. \, \phi) \in FOL(\Sigma, X)$ or $(\forall x. \, \phi) \in FOL(\Sigma, X)$, all free occurrences of x in ϕ are now *bound*. The *free variables of a formula* $\phi \in FOL(\Sigma, X)$ are the variables of ϕ which occur free at least once in ϕ. ■

It is immediate that $QFFOL(\Sigma, X) \subseteq FOL(\Sigma, X)$. Furthermore the variables in a quantifier-free formula ϕ are the free variables of ϕ, and all their occurrences are free in ϕ.

Substitution in a FOL formula is motivated by the need to syntactically express the effect of assignment statements on the formulas of our calculus.

Definition 3.9 (Substitution in a first-order logic formula)**.** Let $X = \langle X_s | s \in S \rangle$ be a family of sets of variable symbols. Let $\phi \in FOL(\Sigma, X)$ a FOL formula. Let $x \in X_s$ a variable symbol of sort $s \in S$. Let $t \in T(\Sigma, X)_s$ a term of sort s. The *substitution* of the variable x in formula ϕ by t, denoted $\phi[x/t]$ is a FOL formula $\chi \in FOL(\Sigma, X)$ such that all free occurrences of x in ϕ are replaced by t. ■

Although the syntactic aspect of formulas has been mentioned prior in this section, evaluation of formulas is required for the semantic aspect dual to the syntactic aspect, notably to determine the assignments that satisfy the formula*. FOL formulas are evaluated to boolean values.

Definition 3.10 (Evaluation of a formula)**.** Let $\Sigma = (S, OP)$ a signature such that $\Sigma_\lambda = \{true, false\}$. Let $X = \langle X_s | s \in S\rangle$ be a family of sets of variable symbols. Let $M \in Mod(\Sigma)$ a Σ-structure such that $true_M = tt$ and $false_M = ff$. Let $\alpha \in [X \to M]$ be an assignment. The *evaluation function of a formula* $eval_{M,\alpha} : FOL(\Sigma, X) \to \{tt, ff\}$ is defined inductively as followed

(i) $eval_{M,\alpha}(\phi) = \phi_M$ for any $\phi \in \Sigma_\lambda$

(ii) $eval_{M,\alpha}(r(t_1, \dots, t_n)) = r_M(eval^{s_1}_{M,\alpha}(t_1), \dots, eval^{s_n}_{M,\alpha}(t_n))$ for all $r \in \Sigma_w$, all non-empty words $w = s_1 ... s_n$, all terms $t_i \in T(\Sigma, X)_{s_i}$ and $i = 1, \dots, n$

(iii) For all $\phi, \chi \in FOL(\Sigma, X)$ and any $x \in X_s$, $s \in S$

(a) $$eval_{M,\alpha}(\neg\phi) = \begin{cases} tt \ if \ eval_{M,\alpha}(\phi) = ff, \\ ff \ otherwise. \end{cases}$$

(b) $$eval_{M,\alpha}(\phi \wedge \chi) = \begin{cases} tt \ if \ eval_{M,\alpha}(\phi) = tt \ and \\ eval_{M,\alpha}(\chi) = tt, \\ ff \ otherwise. \end{cases}$$

(c) $$eval_{M,\alpha}(\phi \vee \chi) = \begin{cases} tt \ if \ eval_{M,\alpha}(\phi) = tt \ or \\ eval_{M,\alpha}(\chi) = tt, \\ ff \ otherwise. \end{cases}$$

(d) $$eval_{M,\alpha}(\exists x.\phi) = \begin{cases} tt \ if \ \exists a \in \Sigma_{\lambda,s}, eval_{M,\alpha}(\phi[x/a]) = tt \\ ff \ otherwise. \end{cases}$$

(e) $$eval_{M,\alpha}(\forall x.\phi) = \begin{cases} tt \ if \ \forall a \in \Sigma_{\lambda,s}, eval_{M,\alpha}(\phi[x/a]) = tt \\ ff \ otherwise. \end{cases}$$

We denote $eval_{M,\alpha}$ the family of evaluation functions for terms and FOL formulas. ■

*Assignments for which the formula is true. We formally define satisfiability of formulas by assignments later in the work.

3.2.4 Required signature for graphs

Prior to the introduction of graphs in the next section, we hereafter set a minimal signature for the graphs to build on. Note that all sorts must be able to express equality, achieved for the int sort with $\leq$.

Definition 3.11 ($\mathcal{Z}_{div}$)**.** The ordered ring of integers with integer division $\mathcal{Z}_{div}$ is an extension of the ring of integers:

$\mathcal{Z}_{div} =$

$$\begin{aligned}
S &= \{int\} \\
\Sigma_{\lambda} &= \{true, false\} \\
\Sigma_{\lambda,int} &= \{0\} \\
\Sigma_{int,int} &= \{SUCC, -\} \\
\Sigma_{int\ int,int} &= \{+, *, div\} \\
\Sigma_{int\ int} &= \{\leq\}
\end{aligned}$$

■

Example 3.2.1. The following structure M is an example of $\mathcal{Z}_{div}$-structure:

$$\begin{aligned}
M_{int} &= \{\dots, -2, -1, 0, 1, 2, \dots\} \\
-_M(x) &= -x \\
SUCC_M(x) &= x + 1 \\
+_M(x, y) &= x + y \\
*_M(x, y) &= x \cdot y \\
div_M(x, y) &= x/y \\
true_M &= tt \\
false_M &= ff \\
\leq_M(x, y) &= \begin{cases} tt\ if\ x \leq y, \\ ff\ otherwise. \end{cases}
\end{aligned}$$

Note that x/y is the quotient of the Euclidean division which we assume is defined on $y = 0$. ■

In this work we introduce the concept of software errors through the generic division by 0 error, in our case $div(x, t)$ such that $eval_{M,\alpha}(t) = 0_M$. There are 3 levels that may deal with the error, in order:

(i) The data type level: in this case $eval_{M,\alpha}(div(x,t)) = \kappa$ with κ some value in the carrier of M for the int sort. This is the design choice of the IEEE 754-2019 standard [35] which by default proposes the $\pm infinity$ value, although for floating-point division. Another possibility is to use an $error$ value in the carrier, by extending the signature with an error symbol in the int sort.

(ii) The language's runtime: for instance the Java programming language throws the *ArithmeticException* when division by 0 is encountered with integer division. Depending on the context (for example if the exception was thrown in a try-catch block) the runtime will decide the next step, possibly crashing the program.

(iii) The operating system: for instance, the Linux operating system will generate a SIGFPE signal and send it to the process that caused the error. The process may handle the signal using a trap, a similar concept to Java's catch blocks, or let the default action to terminate the process occur [36].

If one level does not handle the error, the responsibility of handling the error falls down to the next level. Our work models error handling at the data type level. Modifications of the inference rules and possibly of the graph structure will be necessary to adapt the logic to handle errors at the language's runtime or operating system levels. Conjectures for the modifications will be presented in the discussions.*

Definition 3.12 (Expansion of a signature)**.** Let $\Sigma^1 = (S_1, \Sigma_1)$ and $\Sigma^2 = (S_2, \Sigma_2)$. Σ^2 is an *expansion* of Σ^1, noted $\Sigma^1 \subseteq \Sigma^2$, iff

(i) $S_1 \subseteq S_2$

(ii) for each $s \in S_1$ and each $w \in S_1^*$, $\Sigma^1_{w,s} \subseteq \Sigma^2_{w,s}$

(iii) for each $w \in S_1^*$, $\Sigma^1_w \subseteq \Sigma^2_w$

■

The expansion of a signature permits to extend another signature with additional sorts, operation symbols and relation symbols. We thereafter

*One other approach we have not discussed here is to set special error variables that are different from all program variables. This method also requires modifications to the inference rules, although we conjecture the modifications are less complex to set up than the modifications for the other levels.

require that any graph builds on an expansion of the minimal signature $\mathcal{Z}_{div}$ and has an equality relation symbol for all sorts. It is left to the reader to realise the expansion with any additional sort, constant, operation and relation symbol as they deem fit.

Definition 3.13 (Signatures with integer data type and first-order logic)**.** A *signature with integer data type and FOL formulas* is an expansion

$$\Sigma \supseteq \mathcal{Z}_{div}$$

such that for all $s \in S$, there exists the relation symbol $= \in \Sigma_{s\ s}$ or $\leq \in \Sigma_{s\ s}$. ■

In the sequel, we let:

(i) $\Sigma \supseteq \mathcal{Z}_{div}$ a signature with integer data type and FOL formulas.

(ii) $X = \langle X_s | s \in S \rangle$ a family of sets of variable symbols

(iii) $M \in Mod(\Sigma)$ a many-sorted first-order Σ-structure such that $true_M = tt$ and $false_M = ff$

(iv) $[X \to M]$ the set of all assignments w.r.t. M and X

(v) $eval_{M,\alpha}$ the family of evaluation mappings for terms and FOL formulas for $\alpha \in [X \to M]$

3.3 Control Flow Graphs

Programs can be modelled as graphs upon the basis of the previous section. In this section, we define the basic blocks constituting graphs, precise their structure and elements, as well as define how to execute the graph.

3.3.1 Well-formed graphs

While modern languages have large number of statements, they can be simplified to the assignment and the conditional statements. More complex statements, such as the while-loop, can be expressed by the combination of conditional statements and control flows in the graph.

Definition 3.14 (Statements)**.** The set of (programming) *statements* $Stmt(\Sigma, X)$ over Σ and X is the union of the set $Assign(\Sigma, X)$ of *assignment statements* and the set $Cond(\Sigma, X)$ of *conditional statements* defined as follows

(i) $Assign(\Sigma, X) = \{Assign(\Sigma, X)_s | s \in S\}$ with $Assign(\Sigma, X)_s = \{x := t | x \in X_s, t \in T(\Sigma, X)_s\}$ for any $s \in S$

(ii) $Cond(\Sigma, X) = \{cond | cond \in QFFOL(\Sigma, X)\}$

where all terms t and all first-order formulas $cond$ are finite. ■

Definition 3.15 (Control flow graph)**.** A Control Flow Graph (CFG) over signature Σ and the variables in X is a labelled directed graph defined as follows:

$$G = < V,\ E \subseteq V \times V,\ L_V : V \to Stmt(\Sigma, X),$$
$$L_E : E \to \{true, false\},\ v_0 \in V >$$

where:

(i) V is the set of the vertices in G

(ii) E is the set of the edges in G

(iii) L_V is a mapping of vertices to statements in $Stmt(\Sigma, X)$

(iv) L_E is a mapping of edges to two possible types of exit

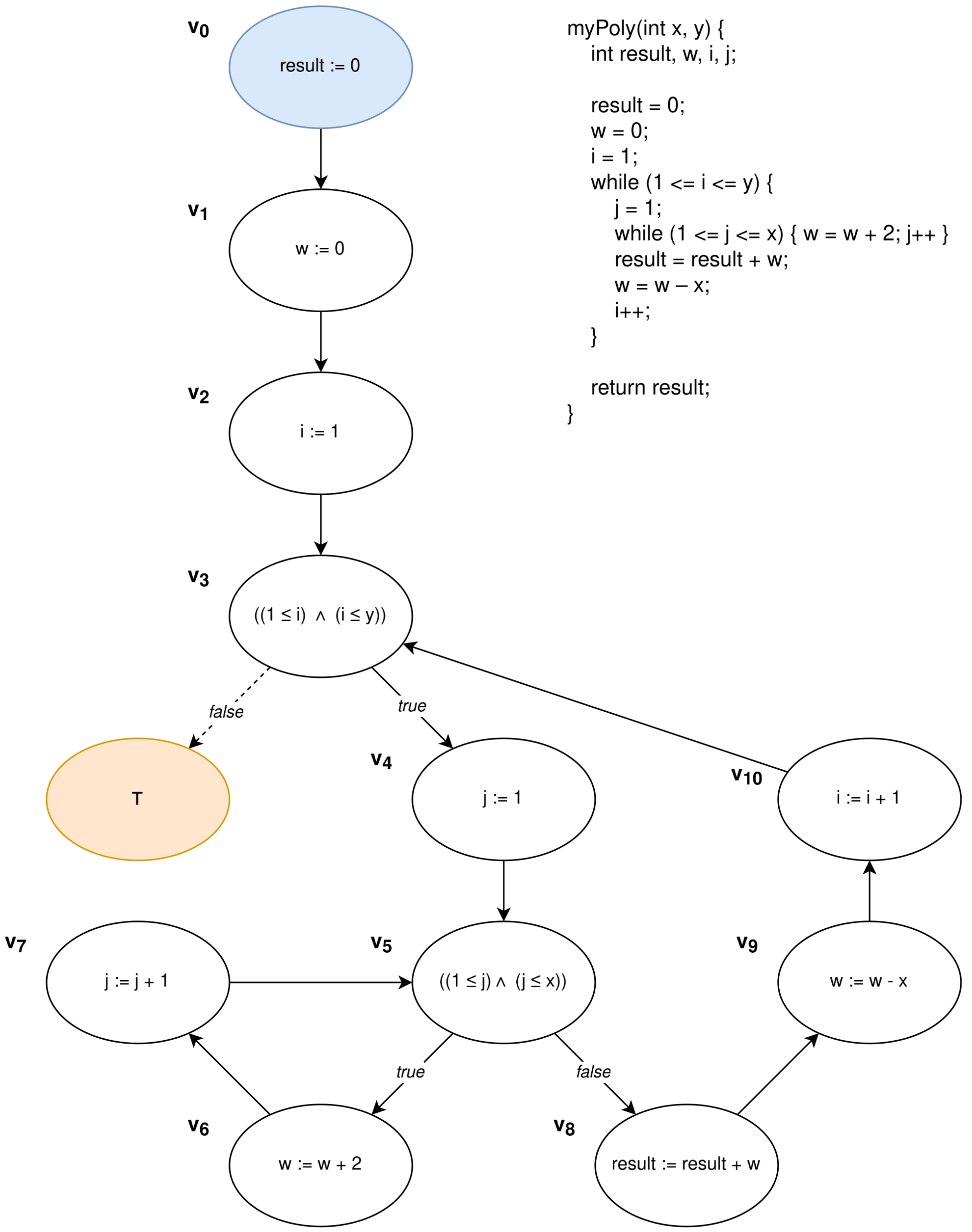
(v) v_0 is the entry vertex in G

■

Example 3.3.1. Fig. 3.1 presents an example of CFG over a signature with integer data type and FOL formulas. Program *myPoly* from [37] is meant to compute $2 * x_0 * y_0$, with x_0, y_0 the binding of x and y in the initial assignment $\alpha_0 \in [X \to M]$, $[X \to M]$ the set of all assignments w.r.t. M and X*. Testing the graph may unravel an error in node v_{10}, where w is not bound to 0 as the correct program would. We will use the graph of Fig. 3.1 as a running example throughout this work. ■

Example 3.3.2. You may note that in Fig. 3.1 we provide program *myPoly* as pseudocode, in the style of the C programming language. Our formal language defined in Section 3.2 translates into programming statements as defined in Definition 3.14. Thus, there is no standard way of writing pseudocode for our programs. We present in the following an example of a Backus-Naur form notation for programs in our language:

*The result is only considered if $x_0, y_0 \geq 0$.



```
myPoly(int x, y) {
    int result, w, i, j;

    result = 0;
    w = 0;
    i = 1;
    while (1 <= i <= y) {
        j = 1;
        while (1 <= j <= x) { w = w + 2; j++ }
        result = result + w;
        w = w – x;
        i++;
    }

    return result;
}
```

Figure 3.1: Program *myPoly* and its CFG

```
<program> ::= (<assignment>
                | <conditional>)
                [";" <program>]

<assignment> ::= <variable> ":=" <term>

<conditional> ::= <FOL> "{"
                            <program>
                            ["|" <program>]
                        "}"
```

■

Definition 3.15 lacks structure constraints for CFGs. For example assignment statements could have more than one exit, and conditional statements more than two. Therefore we define well-formed graphs, and we will only work on them for the remaining of the work.

Definition 3.16 (Well-formed CFG)**.** A CFG G is *well-formed* iff

(i) A node v containing an assignment (i.e. $L_V(v) \in Assign(\Sigma, X)$) has at most one exit edge to a node v' so that $(v, v') \in E$. If there is no v' so that $(v, v') \in E$, then v is called an *exit node*;

(ii) A node v containing a conditional (i.e. $L_V(v) \in Cond(\Sigma, X)$) has at most one exit edge to a node v' (i.e. $(v, v') \in E$) such that $L_E(v, v') = true$ and at most one exit edge to node a v'' (i.e. $(v, v'') \in E$) such that $L_E(v, v'') = false$, and has at least one of the two exits. If v has 1 exit edge, it is an *exit node*.

   (a) $L_E(v, v') = true$ is called the *true exit*

   (b) $L_E(v, v'') = false$ called the *false exit*

We denote $CFG(\Sigma, X)$ the set of all well-formed CFGs over Σ and X. ■

Note that Definition 3.16 allows *goto*-like control flows as there is no restriction to which node the exit points to.

Definition 3.17 (Termination pseudo-node)**.** The *termination pseudo-node* or *termination virtual node* τ is such that when τ is reached by executing the program, the program has terminated. ■

τ does not specify how the program terminated.

3.3.2 Paths and matching

Glass box testing is about executing all paths of a given set. We thereafter define paths in our model, as well as how a path matches a sequence, notably the sequence of nodes that results from the execution of a graph.

Definition 3.18 (Path)**.** Let $G \in CFG(\Sigma, X)$ a well-formed CFG. A *path* p in G is a finite list of at least two nodes in V or ending in τ

$$p = (p_1, \dots, p_n)$$

such that p is *connected*, i.e. $\forall n \in \{1, \dots, n-1\}, (p_n, p_{n+1}) \in E$.

A path is *elementary* if its first node p_1 is v_0.

We denote $Path(G)$ the set of paths of G. ■

Our definition of paths differs from the literature by allowing program termination as a part of it.

Example 3.3.3. In the graph of Fig. 3.1, the following is a non-exhaustive list of paths:

(i) $p_1 = (v_0, v_1, v_2, v_3)$

(ii) $p_2 = (v_3, v_4, v_5, v_8, v_9, v_{10}, v_3)$ (a *loop*, beginning and ending by the same node)

(iii) $p_3 = (v_5, v_6, v_7, v_5, v_6)$ (implying that (v_5, v_6, v_7) is taken at least twice when executing the graph)

(iv) $p_4 = (v_2, v_3, v_4, v_5, v_6, v_7, v_5, v_8)$

■

Definition 3.19 (Loop)**.** Let $G \in CFG(\Sigma, X)$ a well-formed CFG. A *loop* in G is a path $p = (p_1, \dots, p_n) \in Path(G)$ such that $p_1 = p_n$. ■

Definition 3.20 (Path Concatenation)**.** Let $p = (p_1, \dots, p_n)$ and $q = (q_1, \dots, q_m)$ two paths on G such that $(p_n, q_1) \in E$. The *concatenation* of p and q, denoted $p \cdot q$, is the path

$$p \cdot q = (p_1, \dots, p_n, q_1, \dots, q_m)$$

■

Note that the definition implies $p_n \in V$ so $p_n \neq \tau$ and thus only q_m may be τ.

The goal of a glass box test case is to execute a certain path in the program, modelled as a graph in this work. We introduce path matching to assert that the path concerned by the test case is part of the execution sequence of nodes of the test case.

Definition 3.21 (Path matching)**.** Let $G \in CFG(\Sigma, X)$ a well-formed CFG. Let $p = (p_1, \dots, p_n)$ a path in G. p *matches* in an infinite $\mathbb{N}$-indexed sequence $z \in (V \cup \{\tau\})^w$ at $k \in \mathbb{N}$ iff

$$z_k = p_1, \dots, z_{k+n-1} = p_n$$

We say that p matches z if there exists some $k \in \mathbb{N}$ such that p matches z at k, and that p does not match z if there is no such k. ■

To our knowledge, the literature does not specify whether a path is covered when entering or exiting its last node. Ammann and Offut [7], upon whose informal model we base ourselves, does not answer this question. Instead, the authors consider a test case satisfies a test requirement if the execution sequence (referred to in their work as a *test path*) from v_0 to an exit node by the test case contains the test requirement, i.e. the test requirement is a subpath of the test path.

Example 3.3.4. Suppose we have the sequence

$$z = v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_5, v_6, v_7, v_5, v_8, v_9, v_{10}, v_3, \tau, \tau, \dots$$

With v_0 at 0, and only τ in the part that is omitted. Then path $p_1 = (v_0, v_1, v_2, v_3)$ from Example 3.3.3 matches z exactly once at 0, $p_2 = (v_3, v_4, v_5, v_8, v_9, v_{10}, v_3)$ and $p_4 = (v_2, v_3, v_4, v_5, v_6, v_7, v_5, v_8)$ do not match z, but $p_3 = (v_5, v_6, v_7, v_5, v_6)$ matches z once at 5 and another path $p_5 = (v_5, v_6, v_7)$ matches z twice, at 5 and 8. ■

3.3.3 Graph execution

With all the above we can now define the *execution* of a program. This is first defined as an execution step with Definition 3.22.

Definition 3.22 (State transition function δ_G)**.** The *state transition function* on a graph $G \in CFG(\Sigma, X)$, denoted δ_G

$$\delta_G : V \cup \{\tau\} \times [X \to M] \to V \cup \{\tau\} \times [X \to M]$$

is defined by

(i) $\delta_G(\tau, \alpha) = (\tau, \alpha)$ [Convergence]

(ii) if $L_V(v) = x := t$ for $x \in X_s$, $t \in T(\Sigma, X)_s$ and $s \in S$

  (a) if there exists $v' \in V$ such that $(v, v') \in E$, then $\delta_G(v, \alpha) = (v', \alpha[x/eval^s_{M,\alpha}(t)])$ [Assign]

  (b) otherwise v is an exit node and $\delta_G(v, \alpha) = (\tau, \alpha[x/eval^s_{M,\alpha}(t)])$ [Assign-exit]

(iii) if $L_V(v) = cond$ for $cond \in QFFOL(\Sigma, X)$

  (a) if $eval_{M,\alpha}(cond) = tt$ then

    (1) if there exists $v' \in V$ such that $(v, v') \in E$ and $L_E(v, v') = true$, then $\delta_G(v, \alpha) = (v', \alpha)$ [Cond-true]

    (2) otherwise v is an exit node and $\delta_G(v, \alpha) = (\tau, \alpha)$ [Cond-true-exit]

  (b) if $eval_{M,\alpha}(cond) = ff$ then

    (1) if there exists $v'' \in V$ such that $(v, v'') \in E$ and $L_E(v, v'') = false$, then $\delta_G(v, \alpha) = (v'', \alpha)$ [Cond-false]

    (2) otherwise v is an exit node and $\delta_G(v, \alpha) = (\tau, \alpha)$ [Cond-false-exit]

We denote $\delta_{G_{node}}(v, \alpha) = \delta_G(v, \alpha)_1$ and $\delta_{G_{state}}(v, \alpha) = \delta_G(v, \alpha)_2$ respectively the resulting node and state of applying δ_G to some node $v \in V$ and some state $\alpha \in [X \to M]$. ■

With Definition 3.22 we precise the program's behaviour at some node and for some assignment (referred to as a *state* in this context), resulting in a new node-state pair. We also define when and how the program terminates.

Note that the resulting node is not executed after a step, but it will be the next to be executed.

The small step semantics of a CFG is a succession of execution of steps as defined in Definition 3.22.

Definition 3.23 (Small step semantics of CFGs)**.** Let $G \in CFG(\Sigma, X)$. The *small step semantics* of G, denoted δ^*_G, is an infinite sequence

$$\delta^*_G : V \cup \{\tau\} \times [X \to M] \to [\mathbb{N} \to V \cup \{\tau\}] \times [\mathbb{N} \to [X \to M]]$$

defined inductively for any $v \in V \cup \{\tau\}$ and any $\alpha \in [X \to M]$ as follows

(i) $\delta^*_G(v, \alpha)(0) = (v, \alpha)$ [SSSEM: Induction base]

(ii) $\delta^*_G(v, \alpha)(n + 1) = \delta_G(\delta^*_G(v, \alpha)(n))$ [SSSEM: Induction step]

$\delta^*_{G_{node}}(v, \alpha) = \delta^*_G(v, \alpha)_1$ is the *execution path* of G starting in node v and state α and $\delta^*_{G_{state}}(v, \alpha) = \delta^*_G(v, \alpha)_2$ is the *execution state sequence* of G starting in node v and state α. ■

Intuitively the small step semantics of G is the sequence of node-state pairs generated by executing G with δ_G, starting with node v and state α. It is in $\delta^*_{G_{node}}(v_0, \alpha)$ that we will search for a match of the path the test case must exert.

It will now be shown that once τ is reached for any assignment, there is no possibility to reach another node in the graph.

Proposition 3.1 (Program convergence)**.** *Let $v \in V$ be a node and $\alpha \in [X \to M]$ an assignment. If $\delta^*_{G_{node}}(v, \alpha)(n) = \tau$ for some $n \in \mathbb{N}$ then $\delta^*_{G_{node}}(v, \alpha)(m) = \tau$ for all $m \geq n$ and any $\alpha \in [X \to M]$.* ■

Proof. By induction on $m - n$. Let $v \in V \cup \{\tau\}$ and $\alpha \in [X \to M]$. Suppose that for some $n \in \mathbb{N}$,

$$\delta^*_{G_{node}}(v, \alpha)(n) = \tau \tag{3.1}$$

Induction base The case $m - n = 0 \iff m = n$, $\delta^*_{G_{node}}(v, \alpha)(m) = \tau$ holds by hypothesis (3.1).

Induction hypothesis Assume for $x = n + 1, n + 2, \ldots, m - 1$, $\delta^*_{G_{node}}(v, \alpha)(x) = \tau$ holds.

Induction step Suppose $m - n > 0 \iff m > n$. By the induction hypothesis $\delta^*_{G_{node}}(v, \alpha)(m - 1) = \tau$ holds. We must show $\delta^*_{G_{node}}(v, \alpha)(m) = \tau$ holds. SSSEM: Induction step of Definition 3.23 gives us

$$\delta^*_G(v, \alpha)(m) = \delta_G(\delta^*_G(v, \alpha)(m - 1))$$

We know that for some $\alpha' \in [X \to M]$, $\delta^*_G(v, \alpha)(m - 1) = (\tau, \alpha')$. Then by Convergence of Definition 3.22

$$\delta^*_G(v, \alpha)(m) = \delta_G(\delta^*_G(v, \alpha)(m - 1)) = \delta_G(\tau, \alpha') = (\tau, \alpha')$$

Thus finally $\delta^*_{G_{node}}(v, \alpha)(m) = \tau$.

□

The use of Convergence of Definition 3.22 also implies that the state does not change, i.e. If $\delta^*_G(v, \alpha)(n) = (\tau, \alpha')$ for some $n \in \mathbb{N}$ then $\delta^*_G(v, \alpha)(m) = (\tau, \alpha')$ for all $m \geq n$, some $\alpha' \in [X \to M]$ and any $\alpha \in [X \to M]$.

The variable symbols in $X = \langle X_s | s \in S \rangle$ are infinite, yet a test is only concerned by a finite number of variables, notably the program's variables.

Definition 3.24 (Program variables)**.** Let $G \in CFG(\Sigma, X)$ a well-formed CFG. We denote $Vars(G)$ the set of variables contained in the statements of G, also referred to as the *variables of* G.

We denote $[Vars(G) \to M]$ the set of assignments restricted to the variables contained in the statements of G.

For any $\phi \in QFFOL(\Sigma, X)$ first-order formula of free variables $x_1, \ldots, x_n$, we may further denote $\phi[x_1, \ldots, x_n]$ if $x_1, \ldots, x_n \in Vars(G)$ ■

3.3.4 Remark on graph creation

In the example of Fig. 3.1, program *myPoly* does not contain any procedure of function call. If it did, there would be two possible methods to translate it into parts of the graph.

The first approach requires the function to be inlined and then modelled as a set of nodes in the graph. However, if the program contains several calls to the same procedure or function, we must answer the question of how to model the calls in the graph. A tentative approach would model these multiple calls as a single set of nodes, and then connect an edge from all calls to some node considered the entry of the procedure or function. This has two issues: first it might create loops in the graph when there is none in the program, and then the node modelling the exit of the procedure or function should have multiple exit edges, exceeding the limits defined in Definition 3.16. As such this naive approach cannot be used. Instead, each call should correspond to identical, but distinct sets of nodes.

The second approach is to encapsulate all functions and procedures into the signature and the structure. Thus a call corresponds to a single assignment statement.

The choice of the approach to use is ultimately left to the reader. The first tests transitive function calls and may change the test requirements as defined by common glass box coverage models. The second encapsulates the calls, but may not always be achievable, depending on the context (e.g. if the code of the function or procedure is not available).

In the sequel, we let:

(i) $G \in CFG(\Sigma, X)$ a well-formed CFG over Σ and X

(ii) $Vars(G) \subset X$ the set of variables of G

(iii) $[Vars(G) \rightarrow M]$ the set of assignments restricted to the variables contained in the statements of G

3.4 Testing requirements

In this section we define the components of tests and motivate the previous definitions in the context of testing.

3.4.1 Requirements and path satisfiability

Definition 3.25 (Glass box test requirement)**.** A *glass box* or *structural test requirement* is a path in G. ■

Notice that since glass box test requirements are paths, we can verify that a certain execution of the program satisfies them, in which case the test requirement matches the execution node sequence, and after exiting the path a certain state is reached.

Definition 3.26 (Path satisfiability by assignments)**.** Let $\alpha, \beta \in [X \rightarrow M]$ two assignments. Let $p = (p_1, \ldots, p_n) \in Path(G)$ and let $p_{n+1} \in V \cup \{\tau\}$ such that $(p_1, \ldots, p_n, p_{n+1}) \in Path(G)$. The *satisfiability* or *coverage relation* $\vDash$ is a 5-place relation between G, a node $v \in V$, α, $(p_1, \ldots, p_n, p_{n+1})$ and β, in symbols:

$$G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$$

iff

(i) there exists some $m \in \mathbb{N}$ such that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(v, \alpha)$ at m (see Definition 3.21) [Satisfiability: Matching]

(ii) for $k = \mu(m)$ such that p matches $\delta^*_{G_{node}}(v, \alpha)$ at k, $\delta^*_{G_{state}}(v, \alpha)(k+n) = \beta$ [Satisfiability: Postcondition]

We say that G covers p, starting from v and state α, such that the next node to be executed is p_{n+1}, leaving in state β.

A path p is *not covered* by G, starting from v and state α, denoted by

$$G, v, \alpha \vDash \neg p, \beta \iff G, v, \alpha \nvDash p, \beta$$

iff $G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$ does not hold. ■

Requiring p_{n+1} is the synchronicity or ”look ahead” problem. We need to know which node we will take next when we restrict which state β we can take.

It is cumbersome to treat satisfiability assignment per assignment, thus we will generalise the concept to formulas. In this endeavour we first introduce the satisfaction of a formula by an assignment, that is, when the formula evaluates to tt for the assignment.

Definition 3.27 (Satisfaction of a first-order logic formula)**.** Let $\alpha \in [X \to M]$ an assignment. Let $\phi \in FOL(\Sigma, X)$ a FOL formula over Σ and X with free variables $x_1, \ldots, x_n$. The *satisfiability relation* $\vDash$ between M and α and ϕ with free variables $x_1, \ldots, x_n$ is a 3-place relation denoted

$$M, \alpha \vDash \phi(x_1, \ldots, x_n)$$

which is true iff $eval_{M,\alpha}(\phi) = tt$. We denote $M, \alpha \nvDash \phi$ iff $eval_{M,\alpha}(\phi) = ff$. ■

It may be noted that $M, \alpha \nvDash \phi \iff M, \alpha \vDash \neg\phi$.

The groundwork is now laid to introduce satisfiability of a path by a formula, which holds if for every assignment satisfying the formula, the satisfaction of the path by the assignment holds.

Definition 3.28 (Path satisfiability by formulas)**.** Let $\phi, \chi \in FOL(\Sigma, X)$ be FOL formulas over Σ and X. Let $x_1, \ldots, x_m$ the free variables of ϕ and let $y_1, \ldots, y_l$ the free variables of χ. Let $p = (p_1, \ldots, p_n) \in Path(G)$ and let $p_{n+1} \in V \cup \{\tau\}$ such that $(p_1, \ldots, p_n, p_{n+1}) \in Path(G)$. The *satisfiability* or *coverage relation* $\vDash$ is a 5-place relation between G, a node $v \in V$, ϕ, $(p_1, \ldots, p_n, p_{n+1})$ and χ, in symbols:

$$G, v, \phi(x_1, \ldots, x_m) \vDash (p_1, \ldots, p_n, p_{n+1}), \chi[y_1, \ldots, y_l]$$

iff for all $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi(x_1, \ldots, x_m)$ there exists $\beta \in [Vars(G) \to M]$ such that

$$G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$$

and $M, \beta \vDash \chi[y_1, \ldots, y_l]$.

We say that G covers p, starting from v with precondition formula [*] ϕ, such that the next node to be executed is p_{n+1} and such that postcondition χ is satisfied.

[*] Defined in Definition 3.39.

A path p is *not covered* by G, starting from v with precondition $\phi \in FOL(\Sigma, X)$, denoted by

$$G, v, \phi \vDash \neg(p_1, \ldots, p_n, p_{n+1}), \chi \iff G, v, \phi \nvDash (p_1, \ldots, p_n, p_{n+1}), \chi$$

iff there is no $\alpha \in [X \to M]$ and $\beta \in [Vars(G) \to M]$ such that $G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$ with $M, \alpha \vDash \phi(x_1, \ldots, x_m)$ and $M, \beta \vDash \chi[y_1, \ldots, y_l]$. ■

β is unique for a given α, as $\delta^*_G(v, \alpha)$ is deterministic.

Together, the match of $(p_1, \ldots, p_n, p_{n+1})$ with Satisfiability: Matching and Satisfiability: Postcondition assure us of the next step to be taken. Suppose $G, v, \phi \vDash p, \chi$, then for any $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$, $\delta^*_G(v, \alpha)$ is of the following pattern:

$$\delta^*_G(v, \alpha) = (v, \alpha), \ldots, (p_1, \ldots), \ldots, (p_n, \ldots), (p_{n+1}, \beta), \ldots$$

with $\beta \in [Vars(G) \to M]$ and $M, \beta \vDash \chi$. Thus we "lock" what will happen after we cover p.

Example 3.4.1. We have seen before that a path may match an infinite sequence more than once, thus it is possible for test cases to match a path several times, e.g. in Fig. 3.1 any initial assignment α_0 with $x \mapsto 2, y \mapsto 1$ will have $\delta^*_{G_{node}}(v_0, \alpha_0) = v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_5, v_6, v_7, v_5, v_8, v_9, v_{10}, v_3, \tau, \ldots,$ which matches path $p = (v_5, v_6, v_7)$ twice.

By Definition 3.26, a path p is covered by $v \in V$ with precondition $\phi \in FOL(\Sigma, X)$ such that postcondition $\phi \in FOL(\Sigma, X)$ is satisfied for the state after the *first match* of $(p_1, \ldots, p_n, p_{n+1})$ in $\delta^*_{G_{node}}(v, \alpha)$ (Satisfiability: Postcondition). Thus there is no ambiguity in the coverage relation as to which match we describe. ■

In the case the postcondition formula is $true$, it is not simply $(p_1, \ldots, p_n)$ that is covered, but $(p_1, \ldots, p_n, p_{n+1})$. This is notably relevant for the precondition calculus we will introduce in Section 3.5.

Proposition 3.2. *Let $v \in V$ a node of G, $p = (p_1, \ldots, p_n, p_{n+1}) \in Path(G)$ such that $p_{n+1} \in V$ a path in G and $\phi \in QFFOL(\Sigma, X)$ a FOL formula, if $G, v, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$ then G covers $(p_1, \ldots, p_n, p_{n+1})$, starting from v and any state $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$, leaving in any state.* ■

Proof. From $G, v, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$ there is that for any $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$ and for any $\beta \in [X \to M]$ (corresponding to the $true$ postcondition formula)

$$G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$$

Then for each α and for any $\gamma \in [X \to M]$ there is some $v' \in V \cup \{\tau\}$ such that $G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}, v'), \gamma$.

Thus G covers $(p_1, \ldots, p_n, p_{n+1})$ starting in v for any α that satisfies the precondition formula ϕ under M, leaving in any state. □

Since the small step semantics is defined for any node-state pair (v, α) and yields a node-state pairs list, the value at any index k can be used as argument for the small step semantics and $\delta^*_G(\delta^*_G(v, \alpha)(k))$ is a sub-list of $\delta^*_G(v, \alpha)$. Conversely, if we compute the small step semantics for (v', β) and we reach a pair (v, α) for which the small step semantics is known, we can glue the latter sequence to that of (v', β). This is the essence of the following proposition.

Proposition 3.3 (Semantic continuation)**.** *Let* $G \in CFG(\Sigma, X)$ *a well-formed CFG. Let* $\alpha, \beta \in [X \to M]$ *be assignments and let* $v, v' \in V$ *nodes of* G*. If for some index* $k \in \mathbb{N}$, $\delta^*_G(v, \alpha)(k) = (v', \beta)$*, then for all* $l \in \mathbb{N}$

$$\delta^*_G(v, \alpha)(k + l) = \delta^*_G(v', \beta)(l)$$

■

Proof. By induction on l. Let $\alpha, \beta \in [X \to M]$ be assignments and let $v, v' \in V$ nodes of G. Suppose for some $k \in \mathbb{N}$

$$\delta^*_G(v, \alpha)(k) = (v', \beta) \tag{3.2}$$

Induction Base We know that $\delta^*_G(v, \alpha)(k + 0) = (v', \beta)$ by Eq. (3.2). By Definition 3.23, $\delta^*_G(v', \beta)(0) = (v', \beta)$.

Induction hypothesis Let $n \in \mathbb{N}$. Assume for $x = 1, \ldots, n$, $\delta^*_G(v, \alpha)(k+x) = \delta^*_G(v', \beta)(x)$.

Induction step We know by the induction hypothesis that $\delta^*_G(v, \alpha)(k + n) = \delta^*_G(v', \beta)(n)$, we must show $\delta^*_G(v, \alpha)(k + n + 1) = \delta^*_G(v', \beta)(n + 1)$. By SSSEM: Induction step $\delta^*_G(v, \alpha)(k + n + 1) = \delta_G(\delta^*_G(v, \alpha)(k + n))$ and $\delta^*_G(v', \beta)(n + 1) = \delta_G(\delta^*_G(v', \beta)(n))$. Since $\delta^*_G(v, \alpha)(k + n) =$

$\delta^*_G(v', \beta)(n)$ and δ_G is deterministic over all node-assignment pairs by Definition 3.22 then

$$\delta^*_G(v, \alpha)(k + n + 1) = \delta^*_G(v', \beta)(n + 1)$$

□

3.4.2 Testing concepts

For the remaining of the section, we introduce common concepts of testing in the model we have developed in the work.

Definition 3.29 (Test case)**.** A *test case* tc on G is an initial assignment $\alpha_0 \in [X \to M]$ at the entry of G, v_0, restricted to the input variables of G. ■

There may be several assignments at the entry of G (node v_0) that can be restricted to a single test case. Other variables are assumed to be rebound by the program, and global variables are part of the input variables of G.

Assignments are mappings of all variables to values in the respective carriers of the structure, but test cases constrain only some variables. Thus there are several assignments that satisfy a certain test case.

Definition 3.30 (Initial assignments for tc)**.** Let tc a test case on G. An *initial assignment* $\alpha_0 \in [X \to M]$ *for* tc, denoted $tc \subseteq \alpha_0$ is such that for all variables $x \in tc$, $tc(x) = \alpha_0(x)$. ■

This definition prevents the problem of uninitialised variables. The mapping of the variables not in tc can be non-deterministically chosen.

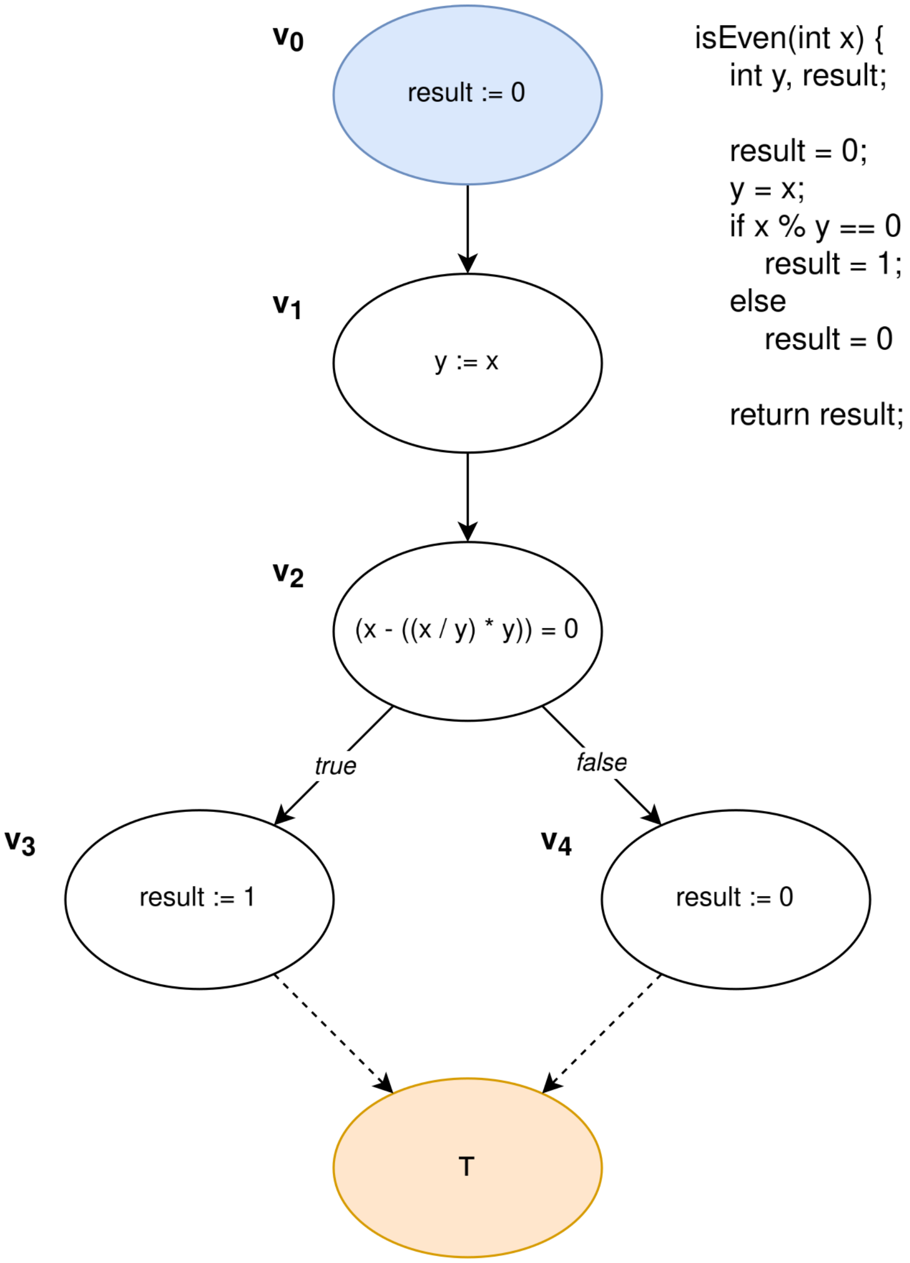
Definition 3.31 (Infeasible path)**.** Let $p \in Path(G)$ a path in G. p is *infeasible* or *dead* iff

$$G, v_0, true \nvDash p, true$$

■

Intuitively, there is no initial assignment that goes down to execute p, as for any initial assignment $\alpha_0 \in [X \to M]$, we have that $M, \alpha_0 \vDash true$ holds. A path that is not infeasible is called feasible or alive.

Example 3.4.2. One such example can be found in Fig. 3.2. Program *isEven* is supposed to answer 1 if input variable x is even, 0 otherwise. A mistake in node v_1 maps y to x instead of 2, so for any x, x mod y is equal to x mod x, that is, to 0. The condition in v_2 which checks whether x modulo y is 0 (remember



```
isEven(int x) {
   int y, result;

   result = 0;
   y = x;
   if x % y == 0
      result = 1;
   else
      result = 0

   return result;
```

Figure 3.2: Program *isEven* and its CFG

that $/$ is the integer division) will always evaluate to tt, and thus edge (v_2, v_4) will never be taken for any initial assignment, (v_2, v_4) is dead. ■

Definition 3.32 (Reachable node)**.** Let $v \in V$ a node. v is *reachable* if there exists a feasible path p on G such that $v \in p$. ■

Conversely, if there is no feasible path containing v, v is unreachable.

Definition 3.33 (Operative edge)**.** Let $(v, v') \in E$. (v, v') is *operative* if there exists at least one feasible path p such that $v, v' \in p$ and v and v' are adjacent in p, in the same order. ■

Definition 3.34 (Satisfiability of path conjunction)**.** Let $v \in V$ a node and let $\phi \in FOL(\Sigma, X)$ a FOL formula. Let $p = (p_1, \ldots, p_n, p_{n+1})$ and $q = (q_1, \ldots, q_m, q_{m+1})$ two paths in G. The *conjunction* of p and q is satisfied by G starting from v with precondition formula ϕ, denoted by

$$G, v, \phi \vDash p \,\&\, q$$

iff $G, v, \phi \vDash p, true$ and $G, v, \phi \vDash q, true$. ■

From negation and conjunction all other Boolean connectives on paths can be defined. In particular, we can define the implication relation:

$$G, v, \phi \vDash p \to q \iff G, v, \phi \nvDash p, true \lor G, v, \phi \vDash q, true.$$

Later in the work we will redefine implication as strong coverage.

Definition 3.35 (Test suite)**.** Let TR a glass box test requirements set on G. A *test suite* TS on G for TR is a set of test cases on G, such that all test requirements $tr \in TR$ are covered by all initial assignments α_0 for at least one test case $tc \in TS$, $\alpha_0 \supseteq tc$. ■

There may be more than one test case that satisfies a given test requirement and a dead path may never be covered.

3.5 Inference rules

In this section we introduce the inference rules of our precondition calculus, and demonstrate some of its properties.

The components of our calculus are inference rules, which under some constraints, the side-conditions, and other provable statements let us infer a proof of another statement.

Definition 3.36 (Inference rule)**.** An *inference rule* is a construct consisting of *premises* $S_1, \dots, S_n$, $n \in \mathbb{N}$, a *conclusion* S and *side conditions* $C_1, \dots, C_m$, $m \in \mathbb{N}$ where $S_1, \dots, S_n, S$ are sequent schemes of the form $G, v, \phi \vdash p, \chi$ or first-order schemes (e.g. $T \vdash \phi \rightarrow \chi$), denoted

$$\text{Name}\,\frac{S_1 \qquad \dots \qquad S_n}{S}\,\begin{array}{c} C_1 \\ \dots \\ C_m \end{array}$$

An *application* of an inference rule is a rule where the premises and the conclusion are replaced by instances of the scheme.

If $n = 0$, the rule is an *axiom*.

A *proof system* is a set of inference rules. ■

In our system the side-conditions are written above the rule, for space concerns.

3.5.1 Precondition calculus

With the groundwork on signatures, graphs and inference rules established, we can now introduce our calculus. It consists of axioms for paths containing only two nodes, and conditional rules to extend a path backwards in the graph, i.e. to a node v that has an edge (v, p_1) to the first node p_1 of the previous path. The Consequence rule (Eq. (3.12)) is used to infer stronger preconditions.

The calculus does not introduce postconditions, contrarily to other calculi in the literature. This is apparent by the use of $true$ on the right-hand side of the sequents. The motivation for this design choice lies in the need for structural testing to generate all preconditions of a path. The postcondition may instead be computed for a specific precondition, and should be treated as an oracle to decide on the test's outcome. We will discuss on postcondition calculi in the future work section.

For each $v \in V$ such that $L_V(v) = x := t$ we have two schema

(i) If $\exists v' \in V$ such that $(v, v') \in E$ then

$$\text{Axiom-Assign}\,\frac{}{G, v, \mathit{true} \vdash (v, v'), \mathit{true}} \tag{3.3}$$

(ii) If $\nexists v' \in V$ such that $(v, v') \in E$ then

$$\text{Axiom-Assign-Exit}\,\frac{}{G, v, \mathit{true} \vdash (v, \tau), \mathit{true}} \tag{3.4}$$

For each $v \in V$ such that $L_V(v) = cond$, we have four schema

(i) (a) If $\exists v' \in V$ such that $(v, v') \in E$ and $L_E(v, v') = true$ then

$$\text{Axiom-Cond-True}\,\frac{}{G, v, cond \vdash (v, v'), true} \tag{3.5}$$

(b) If $\nexists v' \in V$ such that $(v, v') \in E$ and $L_E(v, v') = true$ then

$$\text{Axiom-Cond-True-Exit}\,\frac{}{G, v, cond \vdash (v, \tau), true} \tag{3.6}$$

(ii) (a) If $\exists v' \in V$ such that $(v, v') \in E$ and $L_E(v, v') = false$ then

$$\text{Axiom-Cond-False}\,\frac{}{G, v, (\neg cond) \vdash (v, v'), true} \tag{3.7}$$

(b) If $\nexists v' \in V$ such that $(v, v') \in E$ and $L_E(v, v') = false$ then

$$\text{Axiom-Cond-False-Exit}\,\frac{}{G, v, (\neg cond) \vdash (v, \tau), true} \tag{3.8}$$

For each $v \in V$, each $p = (p_1, \dots, p_n, p_{n+1}) \in Path(G)$ a path in G such that $(v, p_1, \dots, p_n, p_{n+1}) \in Path(G)$

(i) If $L_V(v) = cond$ and $L_E(v, p_1) = true$, then

$$\text{Cond-True}\,\frac{G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true}{G, v, (\phi \wedge cond) \vdash (v, p_1, \dots, p_n, p_{n+1}), true} \tag{3.9}$$

(ii) If $L_V(v) = cond$ and $L_E(v, p_1) = false$, then

$$\text{Cond-False}\,\frac{G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true}{G, v, (\phi \wedge (\neg cond)) \vdash (v, p_1, \dots, p_n, p_{n+1}), true} \tag{3.10}$$

(iii) If $L_V(v) = x := t$, then

$$\text{Assign}\,\frac{G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true}{G, v, \phi[x/t] \vdash (v, p_1, \dots, p_n, p_{n+1}), true} \tag{3.11}$$

Let T a first-order axiomatic theory of the data types of Σ. Let $\phi_1 \in QFFOL(\Sigma, X)$ a finite FOL formula. Then

$$\text{Consequence}\,\frac{T \vdash \phi_1 \rightarrow \phi \qquad G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true}{G, p_1, \phi_1 \vdash (p_1, \dots, p_n, p_{n+1}), true} \tag{3.12}$$

It may be noted that the precondition calculus maintains the well-formedness of formulas, i.e. all ϕ of sequents $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \mathit{true}$ proved by the precondition calculus are finite FOL formulas. The proof by induction on the build-up of the proof tree is left to the reader.

3.5.2 Proof trees and provable statements

Definition 3.37 (Proof tree and provability)**.** Let $\phi, \chi \in QFFOL(\Sigma, X)$ be FOL formulas. Let $v \in V$ a node and let $p = (p_1, \ldots, p_n, p_{n+1}) \in Path(G)$ a path in G. A *proof tree* is a labelled tree where each node is labelled by an instance of the scheme

$$G, v, \phi \vdash p, \chi$$

also referred to as a *sequent* or *sequent inference*, or by an instance of a first-order inference, iff for each node its immediate successors' labels are instances satisfying the premises of an inference rule and the node's label satisfies the same inference rule's conclusion, and all the inference rule's side-conditions hold.

A sequent is *provable* if it is the root node of a proof tree formed by rules for the precondition calculus, i.e. Eqs. (3.3) to (3.12). ■

In essence, a provable sequent is one obtained by recursive proof of its premises, proceeding ”backwards” in the tree. An axiom is always provable if its side-conditions hold. As such all proof trees must contain at least one occurrence of an axiom.

Example 3.5.1. We present several derivations of proof trees for the precondition calculus, with paths from Fig. 3.1. Note that for space reasons we simplify the conditions when obvious (e.g. $1 \leq 1$ is simplified to true and $(\mathit{true} \wedge \phi)$ is simplified to ϕ). We may also show only some nodes in the path, again for space reasons.

Derivation for path $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{G, v_6, true \vdash (v_6, v_7), true}\;\text{Axiom-Assign}}{G, v_5, ((1 \leq j) \wedge (j \leq x)) \vdash (v_5, v_6, v_7), true}\;\text{Cond-True}}{G, v_4, ((1 \leq 1) \wedge (1 \leq x)) \vdash (v_4, v_5, v_6, v_7), true}\;\text{Assign}}{G, v_3, (((1 \leq i) \wedge (i \leq y)) \wedge (1 \leq x)) \vdash (v_3, v_4, v_5, v_6, v_7), true}\;\text{Cond-True}}{G, v_2, (((1 \leq 1) \wedge (1 \leq y)) \wedge (1 \leq x)) \vdash (v_2, v_3, v_4, v_5, v_6, v_7), true}\;\text{Assign}}{G, v_1, ((1 \leq y) \wedge (1 \leq x)) \vdash (v_1, v_2, v_3, v_4, v_5, v_6, v_7), true}\;\text{Assign}}{G, v_0, ((1 \leq y) \wedge (1 \leq x)) \vdash (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7), true}\;\text{Assign}$$

Derivation for path $(v_0, v_1, v_2, v_3, v_4, v_5, v_8)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{G, v_5, (\neg((1 \leq j) \wedge (j \leq x))) \vdash (v_5, v_8), true}\;\text{Axiom-Cond-False}}{G, v_4, (\neg((1 \leq 1) \wedge (1 \leq x))) \vdash (v_4, v_5, v_8), true}\;\text{Assign}}{G, v_3, (((1 \leq i) \wedge (i \leq y)) \wedge (\neg(1 \leq x))) \vdash (v_3, \ldots v_8), true}\;\text{Cond-True}}{G, v_2, (((1 \leq 1) \wedge (1 \leq y)) \wedge (\neg(1 \leq x))) \vdash (v_2, \ldots v_8), true}\;\text{Assign}}{G, v_1, ((1 \leq y) \wedge (\neg(1 \leq x))) \vdash (v_1, v_2, v_3, v_4, v_5, v_8), true}\;\text{Assign}}{G, v_0, ((1 \leq y) \wedge (\neg(1 \leq x))) \vdash (v_0, v_1, v_2, v_3, v_4, v_5, v_8), true}\;\text{Assign}$$

Derivation for path $(v_0, v_1, v_2, v_3, \tau)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{G, v_3, (\neg((1 \leq i) \wedge (i \leq y))) \vdash (v_3, \tau), true}\;\text{Axiom-Cond-False-Exit}}{G, v_2, (\neg((1 \leq 1) \wedge (1 \leq y))) \vdash (v_2, v_3, \tau), true}\;\text{Assign}}{G, v_1, (\neg(1 \leq y)) \vdash (v_1, v_2, v_3, \tau), true}\;\text{Assign}}{G, v_0, (\neg(1 \leq y)) \vdash (v_0, v_1, v_2, v_3, \tau), true}\;\text{Assign}$$

Derivation for path $(v_3, v_4, v_5, v_6, v_7, v_5, v_8)$ where:

(i) $\phi_1 = (\neg((1 \leq j + 1) \wedge (j + 1 \leq x)))$

(ii) $\phi_2 = (\phi_1 \wedge ((1 \leq j) \wedge (j \leq x))) = ((\neg((1 \leq j+1) \wedge (j+1 \leq x))) \wedge ((1 \leq j) \wedge (j \leq x)))$

(iii) $\phi_3 = \phi_2[j/1] = ((\neg((1 \leq 2) \wedge (2 \leq x))) \wedge ((1 \leq 1) \wedge (1 \leq x))) \equiv ((\neg(2 \leq x)) \wedge (1 \leq x))$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{G, v_5, (\neg((1 \leq j) \wedge (j \leq x))) \vdash (v_5, v_8), \mathit{true}}\text{Axiom-Cond-False}}{G, v_7, (\neg((1 \leq j+1) \wedge (j+1 \leq x))) \vdash (v_7, v_5, v_8), \mathit{true}}\text{Assign}}{G, v_6, (\neg((1 \leq j+1) \wedge (j+1 \leq x))) \vdash (v_6, \ldots, v_8), \mathit{true}}\text{Assign}}{G, v_5, (\phi_1 \wedge ((1 \leq j) \wedge (j \leq x))) \vdash (v_5, \ldots, v_8), \mathit{true}}\text{Cond-True}}{G, v_4, \phi_2[j/1] \vdash (v_4, \ldots v_8), \mathit{true}}\text{Assign}}{G, v_3, (\phi_3 \wedge ((1 \leq i) \wedge (i \leq y))) \vdash (v_3, \ldots, v_8), \mathit{true}}\text{Cond-True}$$

and so the final inference is $G, v_3, ((\neg(2 \leq x)) \wedge (1 \leq x)) \wedge ((1 \leq i) \wedge (i \leq y))) \vdash (v_3, v_4, v_5, v_6, v_7, v_5, v_8), \mathit{true}$

■

3.5.3 Soundness

In this section we show that our proof system is *sound*. The soundness property requires that all provable statements are true, i.e. that for any provable statement $G, v, \phi \vdash p, \chi$ then $G, v, \phi \vDash p, \chi$ is true. Note that as corollary, no false statement may be generated by the logic, and as such nor can contradictions.

A notable property of our calculus is that for any provable sequent $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \mathit{true}$ it guarantees any execution starting in p_1 with any assignment $\alpha \in [X \rightarrow M]$ such that $M, \alpha \vDash \phi$ instantly executes $(p_1, \ldots, p_n, p_{n+1})$. The proof is obtained by induction on the build-up of the proof tree. We show that the property holds for the axioms of our calculus. Then we show that if the property holds for a path $(p_1, \ldots, p_n, p_{n+1})$ and the non-axiom rules guarantee the property holds for their resulting path, then the property is guaranteed for all paths with our calculus. With the proof of immediate execution, we obtain that the path matches the execution sequence at 0, and so in general at some $k \in \mathbb{N}$. Thus, the proof of soundness for the calculus becomes trivial.

Lemma 3.1 (Immediate matching with the precondition calculus)**.** *The proof system for precondition calculus formed by rules Eqs.* (3.3) *to* (3.12) *is such that for any provable sequent* $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \mathit{true}$ *if* $M, \alpha \vDash \phi$ *then* $(p_1, \ldots, p_n, p_{n+1})$ *matches* $\delta^*_{G_{node}}(p_1, \alpha)$ *at* 0. ■

Proof. By induction on the build-up of the proof tree.

Base case We treat the axioms of the proof system.

(i) Axiom-Assign (Eq. (3.3): Take $G, v, \mathit{true} \vdash (v, v'), \mathit{true}$ with $v' \in V$. Let $\alpha \in [X \rightarrow M]$ an assignment. As $L_V(v) = x :=$

t by the side conditions of Eq. (3.3), Assign gives us $\delta_G(v, \alpha) = (v', \alpha[x/eval_{M,\alpha}(t)])$. Thus (v, v') matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(ii) Axiom-Assign-Exit (Eq. (3.4)): Take $G, v, true \vdash (v, \tau), true$. Let $\alpha \in [X \rightarrow M]$ an assignment. As $L_V(v) = x := t$ and v is an exit node by the side conditions of Eq. (3.4), Assign-exit gives us $\delta_G(v, \alpha) = (\tau, \alpha[x/eval_{M,\alpha}(t)])$. Thus (v, τ) matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iii) Axiom-Cond-True (Eq. (3.5): Take $G, v, cond \vdash (v, v'), true$. Let $\alpha \in [X \rightarrow M]$ an assignment such that $M, \alpha \models cond$. As $L_V(v) = cond$ and $L_E(v, v') = true$ by the side conditions of Eq. (3.5), Cond-true gives us $\delta_G(v, \alpha) = (v', \alpha)$. Thus (v, v') matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iv) Axiom-Cond-True-Exit (Eq. (3.6): Take $G, v, cond \vdash (v, \tau), true$. Let $\alpha \in [X \rightarrow M]$ an assignment such that $M, \alpha \models cond$. As $L_V(v) = cond$ and $\nexists v' \in V$ such that $L_E(v, v') = true$ by the side conditions of Eq. (3.6), Cond-true-exit gives us $\delta_G(v, \alpha) = (\tau, \alpha)$. Thus (v, τ) matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(v) Axiom-Cond-False (Eq. (3.7): Take $G, v, (\neg cond) \vdash (v, v'), true$. Let $\alpha \in [X \rightarrow M]$ an assignment such that $M, \alpha \models (\neg cond)$. As $L_V(v) = cond$ and $L_E(v, v') = false$ by the side conditions of Eq. (3.7), Cond-false gives us $\delta_G(v, \alpha) = (v', \alpha)$. Thus (v, v') matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(vi) Axiom-Cond-False-Exit (Eq. (3.8): Take $G, v, (\neg cond) \vdash (v, \tau), true$. Let $\alpha \in [X \rightarrow M]$ an assignment such that $M, \alpha \models (\neg cond)$. As $L_V(v) = cond$ and $\nexists v' \in V$ such that $L_E(v, v') = false$ by the side conditions of Eq. (3.8), Cond-false-exit gives us $\delta_G(v, \alpha) = (\tau, \alpha)$. Thus (v, τ) matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

Induction hypothesis Consider the final inference in a proof tree to be a sequent inference of the form $G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true$. Then assume the sequent inferences $G, q_1, \chi \vdash (q_1, \dots, q_m, q_{m+1}), true$ in the premises of the final inference are such that for any $\alpha \in [X \rightarrow M]$, if $M, \alpha \models \chi$ then $(q_1, \dots, q_m, q_{m+1})$ matches $\delta^*_{G_{node}}(q_1, \alpha)$ at 0.

Induction step

(i) Cond-True (Eq. (3.9)): Suppose $G, v, (\phi \wedge cond) \vdash (v, p_1, \dots, p_n, p_{n+1})$, $true$ is provable. By the side-conditions of Eq. (3.9) we have that $L_V(v) = cond$ and $L_E(v, p_1) = true$. Thus for all $\alpha \in [X \rightarrow M]$ such that $M, \alpha \models (\phi \wedge cond)$, by Cond-true $\delta_G(v, \alpha) = (p_1, \alpha)$. By the premise of Eq. (3.9) and the induction hypothesis, we have that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \beta)$ at 0 for any $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$. Since $M \vDash (\phi \wedge cond) \to \phi$ and $M, \alpha \vDash (\phi \wedge cond)$, we have that $M, \alpha \vDash \phi$. Thus, $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. Finally with $\delta_G(v, \alpha) = (p_1, \alpha)$ and Proposition 3.3 we have that $(v, p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{state}}(v, \alpha)$ at 0.

(ii) Cond-False (Eq. (3.10)): Suppose $G, v, (\phi \wedge (\neg cond)) \vdash (v, p_1, \ldots, p_n, p_{n+1}), true$ is provable. By the side-conditions of Eq. (3.10) we have that $L_V(v) = cond$ and $L_E(v, p_1) = false$. Thus for all $\alpha \in [X \to M]$ such that $M, \alpha \vDash (\phi \wedge (\neg cond))$, by Cond-false $\delta_G(v, \alpha) = (p_1, \alpha)$. By the premise of Eq. (3.10) and the induction hypothesis, we have that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \beta)$ at 0 for any $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$. Since $M \vDash (\phi \wedge (\neg cond)) \to \phi$ and $M, \alpha \vDash (\phi \wedge (\neg cond))$, we have that $M, \alpha \vDash \phi$. Thus, $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. Finally with $\delta_G(v, \alpha) = (p_1, \alpha)$ and Proposition 3.3 we have that $(v, p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iii) Assign (Eq. (3.11)): Suppose $G, v, \phi[x/t] \vdash (v, p_1, \ldots, p_n, p_{n+1}), true$ is provable. By the side-conditions of Eq. (3.11) we have that $L_V(v) = x := t$ and $(v, p_1) \in E$. Thus for all $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi[x/t]$, by Assign $\delta_G(v, \alpha) = (p_1, \alpha[x/eval_{M,\alpha}(t)])$. By the premise of Eq. (3.11) and the induction hypothesis, we have that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \beta)$ at 0 for any $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$. Since $M, \alpha \vDash \phi[x/t]$ then $M, \alpha[x/eval_{M,\alpha}(t)] \vDash \phi$. Thus, $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha[x/eval_{M,\alpha}(t)])$ at 0. Finally with $\delta_G(v, \alpha) = (p_1, \alpha[x/eval_{M,\alpha}(t)])$ and Proposition 3.3 we have that $(v, p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iv) Consequence (Eq. (3.12)): Suppose $G, p_1, \phi_1 \vdash (p_1, \ldots, p_n, p_{n+1}), true$ is provable. By the premises of Eq. (3.12) and the induction hypothesis, we have that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \beta)$ at 0 for any $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$ and $T \vdash \phi_1 \to \phi$. Let $\alpha \in [X \to M]$ an assignment such that $M, \alpha \vDash \phi_1$. Since $T \vdash \phi_1 \to \phi$ then $M \vDash \phi_1 \to \phi$ and since $M, \alpha \vDash \phi_1$ we have that $M, \alpha \vDash \phi$. Thus $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at 0.

□

As a consequence of Lemma 3.1, any provable sequent is such that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ for any $\alpha \in [X \to M]$ such that

$M, \alpha \vDash \phi$ holds as required by Satisfiability: Matching. There is only left to prove that the postcondition holds, which is trivial.

Theorem 3.1 (Soundness of the precondition calculus)**.** *Let* $\phi \in FOL(\Sigma, X)$ *be a FOL formula. Let* $p = (p_1, \dots, p_n, p_{n+1}) \in Path(G)$ *a path in G. The proof system for the precondition calculus formed by rules Eqs.* (3.3) *to* (3.12) *is sound, i.e. for all provable sequents* $G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true$ *we have*

$$G, p_1, \phi \vDash (p_1, \dots, p_n, p_{n+1}), true$$

■

Proof. By Lemma 3.1 we have that for any provable sequent $G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true$ and any $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$, $(p_1, \dots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. Its result can be weakened to $(p_1, \dots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at $k \in \mathbb{N}$. Thus the matching is satisfied.

It is left to prove that the postcondition holds, which is trivial as for any $\beta \in [X \to M]$, there is $M, \beta \vDash true$. Thus $M, \delta^*_{G_{state}}(p_1, \alpha)(k + n) \vDash true$. □

3.5.4 Completeness

We have previously described soundness as proof implies truth. Completeness is achieved when the opposite holds, when truth implies proof, so if $G, v, \phi \vDash p, \chi$ is true then $G, v, \phi \vdash p, \chi$ is provable. It is not the case for our precondition calculus.

Theorem 3.2 (Incompleteness of the precondition calculus)**.** *The proof system for the precondition calculus formed by rules Eqs.* (3.3) *to* (3.12) *is* incomplete, *i.e. there exists some node* $v \in V$*, some path* $p = (p_1, \dots, p_n, p_{n+1}) \in Path(G)$ *and some first-order formulas* $\phi, \chi \in QFFOL(\Sigma, X)$ *such that*

$$G, v, \phi \vDash (p_1, \dots, p_n, p_{n+1}), \chi$$

but

$$G, v, \phi \vdash (p_1, \dots, p_n, p_{n+1}), \chi$$

is not provable. ■

Proof. By counter-example. By Theorem 3.1 for some sequent $G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true$ we have $G, p_1, \phi \vDash (p_1, \dots, p_n, p_{n+1}), true$.

But then $G, p_1, \phi \vDash (p_2, \dots, p_n, p_{n+1}), true$ is also true, while the precondition calculus rules do not permit to prove any sequence $G, v, \phi \vDash$

$(p_1, \dots, p_n, p_{n+1})$ where $v \neq p_1$, thus we cannot prove $G, p_1, \phi \vdash (p_2, \dots, p_n, p_{n+1}), true$. □

Later in the work, we will demonstrate the precondition calculus is complete w.r.t. certain truths under constraints on the form of the graph. The proof requires definitions and theorems that have yet to be introduced.

3.6 Weakest precondition, coverage and calculus properties

In this section we define weakest preconditions and weakest precondition formulas, as well as notably prove our calculus can obtain a weakest precondition formula for a path under some restrictions.

3.6.1 Coverage and relations

Definition 3.38 (Weakest preconditions)**.** Let $p \in Path(G)$ a path. Let $v \in V$ a node of G. The *weakest precondition* for p starting in v, denoted $WP_{p,v}$ is:

$$WP_{p,v} = \{\alpha \in [X \to M] | \exists \beta \in [X \to M].\, G, v, \alpha \vDash p, \beta\}$$

Conversely, an assignment $\gamma \in [X \to M]$ such that $\gamma \notin WP_{p,v}$ implies that for any $\beta \in [X \to M]$, $G, v, \gamma \nvDash p, \beta$.

A set of assignments Γ is a *precondition* of p starting in v iff $\Gamma \subseteq WP_{p,v}$. ■

In particular, WP_{p,v_0} is the set of all initial assignments for all test cases of p.

We now define weakest precondition formulas as formulas that are satisfied by all and only the assignments in the corresponding weakest precondition. The weakest precondition formulas and precondition formulas can be used with our calculus.

Definition 3.39 (Weakest precondition formulas)**.** Let $\phi \in QFFOL(\Sigma, X)$ a formula. Let $p \in Path(G)$ a path in G. Let $v \in V$ a node of G. ϕ is a *weakest precondition formula* for p starting in v iff the set Γ of all assignments $\alpha \in \Gamma$ such that $M, \alpha \vDash \phi$ is $WP_{p,v}$, i.e. $\Gamma = WP_{p,v}$.

A formula $\chi \in QFFOL(\Sigma, X)$ is a *precondition formula* for p starting in v iff for all $\beta \in [X \to M]$ such that $M, \beta \vDash \chi$ then $\beta \in WP_{p,v}$. ■

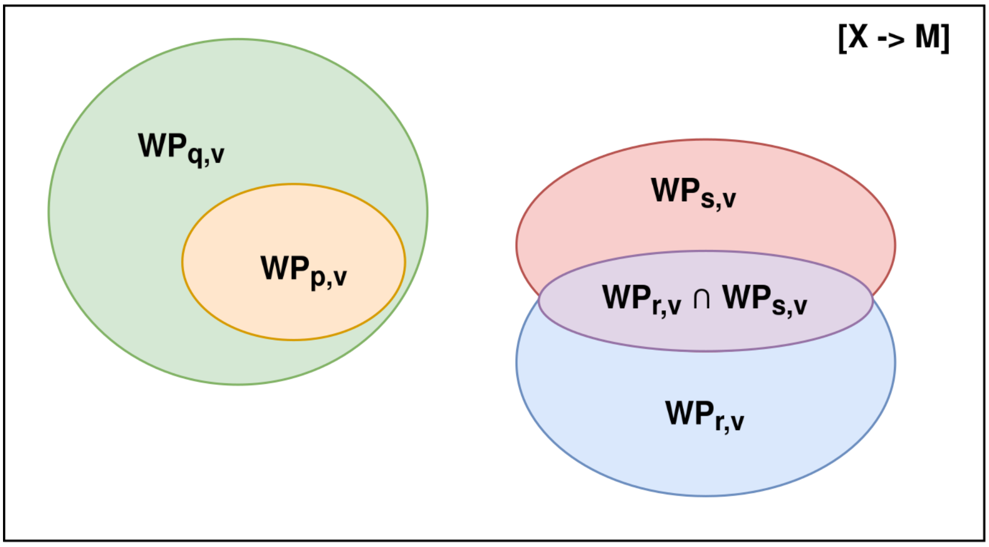


Figure 3.3: Graphical representation of weakest preconditions

It may be noted that χ being a precondition formula for p starting in v is equivalent to

$$G, v, \chi \vDash p, true$$

The weakest precondition for a path p starting in a node v is a superset of all preconditions for p starting in v. This implies that precondition formulas for p starting in v are stricter than the weakest precondition formulas for p starting in v, as expressed by the following proposition.

Proposition 3.4. *Let $p \in Path(G)$ a path. Let $v \in V$ a node of G. If there exists a weakest precondition formula $\phi \in QFFOL(\Sigma, X)$ for p starting in v then for any precondition formula χ for p starting in v then $M \vDash \chi \to \phi$, i.e. for any $\alpha \in [X \to M]$*

$$M, \alpha \vDash \chi \implies M, \alpha \vDash \phi$$

■

Proof. Suppose ϕ is a weakest precondition formula for p starting in v and χ any precondition formula for p starting in v. Then for all $\alpha \in [X \to M]$ such that $M, \alpha \vDash \chi$, as $\alpha \in WP_{p,v}$ we also have $M, \alpha \vDash \phi$. □

Example 3.6.1. Fig. 3.3 presents a graphical representation of weakest preconditions for paths p, q, r, s starting in $v \in V$ and their relations. $WP_{p,v} \subseteq WP_{q,v}$, so any initial assignment in $WP_{p,v}$ will satisfy both p and q starting in

v so all test cases of p are test cases of q. $WP_{r,v} \cap WP_{s,v} \neq \varnothing$, so some test cases for r also cover s starting in v. Finally, $WP_{p,v} \cap WP_{r,v} = \varnothing$ thus no test case for p can cover r starting in v. ■

We now introduce relations on the weakest preconditions for different paths starting in v_0.

Definition 3.40 (Strong requirement coverage)**.** Let p and q reachable paths in G. Let WP_{p,v_0} the weakest precondition for p starting in v_0 and let WP_{q,v_0} the weakest precondition for q starting in v_0. p *strongly covers* q, noted $p \Rightarrow q$ iff $WP_{p,v_0} \subseteq WP_{q,v_0}$. ■

From the preceding definition, if $p \Rightarrow q$ then any test case for p is a test case for q, and q may be safely removed from the requirements set. This in turn reduces the number of test cases in a test suite.

Definition 3.41 (Weak requirement coverage)**.** Let p and q reachable paths in G. Let WP_{p,v_0} the weakest precondition for p starting in v_0 and let WP_{q,v_0} the weakest precondition for q starting in v_0. p *weakly covers* q, noted $p \rightarrow q$ iff $WP_{p,v_0} \cap WP_{q,v_0} \neq \varnothing$. Weak coverage is commutative. ■

From the preceding definition, if $p \rightarrow q$ then some test cases for p are test cases for q, and one such common test case can be selected to satisfy both paths.

Definition 3.42 (Requirement independence)**.** Let p and q reachable paths in G. Let WP_{p,v_0} the weakest precondition for p starting in v_0 and let WP_{q,v_0} the weakest precondition for q starting in v_0. p is *independent* of q, noted $p \nsim q$ iff $WP_{p,v_0} \cap WP_{q,v_0} = \varnothing$. Independence is commutative. ■

From the preceding definition, if $p \nsim q$ then there is no test case for p that is also a test case for q, and at least two test cases are required to satisfy both paths.

Proposition 3.5 (Strong requirement coverage with formulas)**.** *Let $\phi, \chi \in QFFOL(\Sigma, X)$ be formulas. Let $p, q \in Path(G)$ paths in G. Let $v \in V$ a node of G. If ϕ is a weakest precondition formula for p starting in v and χ is a weakest precondition formula for q starting in v and $WP_{p,v} \subseteq WP_{q,v}$, then*

$$M \vDash \phi \rightarrow \chi$$

■

Proof. Suppose:

(i) ϕ is a weakest precondition formula for p starting in v

(ii) χ is a weakest precondition formula for q starting in v

(iii) $WP_{p,v} \subseteq WP_{q,v}$

This implies that for all $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$ there is that $\alpha \in WP_{p,v} \implies \alpha \in WP_{q,v}$. Thus $M, \alpha \vDash \chi$, and $M \vDash \phi \to \chi$. □

3.6.2 Precondition calculus and weakest preconditions

We can deduce from soundness that the formulas obtained by our calculus for a path $p = (p_1, \ldots, p_n)$ are precondition formulas for p starting in p_1.

Corollary 3.1. *Theorem 3.1 implies that for any provable sequent* $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ *there is that* ϕ *is a precondition formula for* $(p_i, \ldots, p_j)$*,* $1 \leq i < j \leq n + 1$ *starting in* p_1*.* ■

Proof. Because the system is sound, $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1})$ being provable implies that

$$G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$$

which fits the definition of a precondition formula for $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1.

Since $G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$, for any $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$ we have that $(p_1, \ldots, p_n, p_{n+1})$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at some $k \in \mathbb{N}$. Thus $(p_i, \ldots, p_j)$ matches $\delta^*_{G_{node}}(p_1, \alpha)$ at $k + i - 1$. Finally $M, \delta^*_{G_{state}}(p_1, \alpha)(k + j - 1) \vDash true$, so

$$G, p_1, \phi \vDash (p_i, \ldots, p_j), true$$

and thus ϕ is a precondition formula for $(p_i, \ldots, p_j)$, $1 \leq i < j \leq n + 1$ starting in p_1. □

Note this only establishes ϕ is a precondition formula for $(p_i, \ldots, p_j)$, not a weakest precondition formula.

A notable property of our calculus is that the precondition formulas obtained must be satisfied at some point in any execution sequence for the path to be taken, i.e. for a precondition formula ϕ for $(p_1, \ldots, p_n, p_{n+1})$ a pair (p_1, α) with $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$ must be reached to execute $(p_1, \ldots, p_n, p_{n+1})$. Furthermore, when taking into account the result of Lemma 3.1 the path is executed immediately when this condition is satisfied.

Lemma 3.2 (Necessary precondition by the precondition calculus)**.** *The proof system for the precondition calculus formed by rules Eqs.* (3.3) *to* (3.12) *yields necessary preconditions for paths without using Eq.* (3.12)*, i.e. for a provable sequent* $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ *where the proof tree does not contain the Consequence rule (Eq.* (3.12)*),* ϕ *is such that:*

(i) ϕ *is a precondition formula for* $(p_1, \ldots, p_n, p_{n+1})$ *starting in* p_1

(ii) if for any $\alpha \in [X \to M]$ *such that* $M, \alpha \nvDash \phi$ *then* $(p_1, \ldots, p_n, p_{n+1})$ *does not match* $\delta^*_{G_{node}}(p_1, \alpha)$ *at* 0.

■

Proof. ϕ being a precondition formula for $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1 is given by Corollary 3.1 for any provable sequent $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$.

There is left to prove that for any $\alpha \in [X \to M]$ if $M, \alpha \nvDash \phi$ then $(p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. We prove so by induction on the build-up of the proof tree, that does not contain the Consequence rule (Eq. (3.12)).

Induction Base We treat the axioms of the proof system.

(i) Axiom-Assign (Eq. (3.3)): Take $G, v, true \vdash (v, v'), true$ with $v' \in V$. As $M, \beta \vDash true$ for any $\beta \in [X \to M]$, there is no $\alpha \in [X \to M]$ such that $M, \alpha \nvDash true$. Thus the case is closed.

(ii) Axiom-Assign-Exit (Eq. (3.4)): Take $G, v, true \vdash (v, \tau), true$. As $M, \beta \vDash true$ for any $\beta \in [X \to M]$, there is no $\alpha \in [X \to M]$ such that $M, \alpha \nvDash true$. Thus the case is closed.

(iii) Axiom-Cond-True (Eq. (3.5)): Take $G, v, cond \vdash (v, v'), true$. Let $\alpha \in [X \to M]$ an assignment such that $M, \alpha \nvDash cond$. As $L_V(v) = cond$ and $L_E(v, v') = true$ by the side conditions of Eq. (3.5), then by Definition 3.22 either $\delta_G(v, \alpha) = (v'', \alpha)$ with $L_E(v, v'') = false$ or $\delta_G(v, \alpha) = (\tau, \alpha)$ if there exists no such v''. Thus (v, v') does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iv) Axiom-Cond-True-Exit (Eq. (3.6)): Take $G, v, cond \vdash (v, \tau), true$. Let $\alpha \in [X \to M]$ an assignment such that $M, \alpha \nvDash cond$. As $L_V(v) = cond$ and $\nexists v' \in V$ such that $L_E(v, v') = true$ by the side conditions of Eq. (3.6), and there must be an exit edge $(v, v'') \in E$ such that $L_E(v, v'') = false$ by Definition 3.16, then by Cond-false $\delta_G(v, \alpha) = (v'', \alpha)$. Thus (v, τ) does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(v) Axiom-Cond-False (Eq. (3.7)): Take $G, v, (\neg cond) \vdash (v, v'), true$. Let $\alpha \in [X \to M]$ an assignment such that $M, \alpha \nvDash (\neg cond)$. As $L_V(v) = cond$ and $L_E(v, v') = false$ by the side conditions of Eq. (3.7), then by Definition 3.22 either $\delta_G(v, \alpha) = (v'', \alpha)$ with $L_E(v, v'') = true$ or $\delta_G(v, \alpha) = (\tau, \alpha)$ if there exists no such v''. Thus (v, v') does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(vi) Axiom-Cond-False-Exit (Eq. (3.8)): Take $G, v, (\neg cond) \vdash (v, \tau), true$. Let $\alpha \in [X \to M]$ an assignment such that $M, \alpha \nvDash (\neg cond)$. As $L_V(v) = cond$ and $\nexists v' \in V$ such that $L_E(v, v') = false$ by the side conditions of Eq. (3.8), and there must be an exit edge $(v, v'') \in E$ such that $L_E(v, v'') = true$ by Definition 3.16, then by Cond-true $\delta_G(v, \alpha) = (v'', \alpha)$. Thus (v, τ) does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

Induction hypothesis Consider the final inference in a proof tree that does not contain the Consequence rule (Eq. (3.12)) to be a sequent inference of the form $G, p_1, \phi \vdash (p_1, \dots, p_n, p_{n+1}), true$. Then assume the sequent inferences $G, q_1, \chi \vdash (q_1, \dots, q_m, q_{m+1}), true$ in the premises of the final inference are such that for any $\alpha \in [X \to M]$, if $M, \alpha \nvDash \chi$ then $(q_1, \dots, q_m, q_{m+1})$ does not match $\delta^*_{G_{node}}(q_1, \alpha)$ at 0.

Induction step

(i) Cond-True (Eq. (3.9)): Suppose $G, v, (\phi \wedge cond) \vdash (v, p_1, \dots, p_n, p_{n+1})$, $true$ is provable without using Eq. (3.12). Let $\alpha \in [X \to M]$ such that $M, \alpha \nvDash (\phi \wedge cond)$. By the side-conditions of Eq. (3.9) we have that $L_V(v) = cond$ and $L_E(v, p_1) = true$. From $M, \alpha \nvDash (\phi \wedge cond)$, we have that $M, \alpha \vDash (\neg(\phi \wedge cond)) \iff M, \alpha \vDash ((\neg\phi) \vee (\neg cond)))$.

Case $M, \alpha \nvDash cond$: by Definition 3.22 either $\delta_G(v, \alpha) = (v', \alpha)$ with $L_E(v, v') = false$ or $\delta_G(v, \alpha) = (\tau, \alpha)$ if there is no such v'. Thus $(v, p_1, \dots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

Case $M, \alpha \vDash (cond \wedge (\neg\phi))$: by Cond-true $\delta_G(v, \alpha) = (p_1, \alpha)$. $M, \alpha \vDash (cond \wedge (\neg\phi)) \implies M, \alpha \nvDash \phi$. By the induction hypothesis $(p_1, \dots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. Finally, by Proposition 3.3 $(v, p_1, \dots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(ii) Cond-False (Eq. (3.10)): Suppose $G, v, (\phi \wedge (\neg cond)) \vdash (v, p_1, \ldots, p_n, p_{n+1}), true$ is provable without using Eq. (3.12). Let $\alpha \in [X \to M]$ such that $M, \alpha \nvDash (\phi \wedge (\neg cond))$. By the side-conditions of Eq. (3.10) we have that $L_V(v) = cond$ and $L_E(v, p_1) = false$. Since $M, \alpha \nvDash (\phi \wedge (\neg cond))$ we have that $M, \alpha \vDash (\neg(\phi \wedge (\neg cond))) \iff M, \alpha \vDash ((\neg\phi) \vee cond)$.

Case $M, \alpha \vDash cond$: by Definition 3.22 either $\delta_G(v, \alpha) = (v', \alpha)$ with $L_E(v, v') = true$ or $\delta_G(v, \alpha) = (\tau, \alpha)$ if there is no such v'. Thus $(v, p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

Case $M, \alpha \vDash ((\neg cond) \wedge (\neg\phi))$: by Cond-false $\delta_G(v, \alpha) = (p_1, \alpha)$. $M, \alpha \vDash ((\neg cond) \wedge (\neg\phi)) \implies M, \alpha \nvDash \phi$. By the induction hypothesis $(p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(p_1, \alpha)$ at 0. Finally, by Proposition 3.3 $(v, p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \alpha)$ at 0.

(iii) Assign (Eq. (3.11)): Suppose $G, v, \phi[x/t] \vdash (v, p_1, \ldots, p_n, p_{n+1}), true$ is provable without using Eq. (3.12). By the side-conditions of Eq. (3.11) we have that $L_V(v) = x := t$ and $(v, p_1) \in E$. Thus for any $\alpha \in [X \to M]$, by Assign $\delta_G(v, \alpha) = (p_1, \alpha[x/eval_{M,\alpha}(t)])$. By the premise of Eq. (3.11) and the induction hypothesis, we have that $(p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(p_1, \beta)$ at 0 for any $\beta \in [X \to M]$ such that $M, \beta \nvDash \phi$. Let us take $\gamma \in [X \to M]$ such that $M, \gamma \nvDash \phi[x/t]$. Since $M, \gamma \nvDash \phi[x/t]$, we have that $M, \gamma[x/eval_{M,\gamma}(t)] \nvDash \phi$. Thus, $(p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(p_1, \gamma[x/eval_{M,\alpha}(t)])$ at 0. Finally with $\delta_G(v, \gamma) = (p_1, \gamma[x/eval_{M,\gamma}(t)])$ and Proposition 3.3 we have that $(v, p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \gamma)$ at 0.

□

Lemma 3.2 implies that to take the path we have constructed with the precondition calculus, we must satisfy the precondition at the first node of the path, somewhere in the small step semantics $\delta^*_G(v, \alpha)$ for any $v \in V$ and any $\alpha \in [X \to M]$. Conversely, if ϕ is the precondition for path p obtained by our precondition calculus and if there is never $\delta^*_G(v, \alpha)(k) = (p_1, \beta)$ with $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$ for any $k \in \mathbb{N}$ then $G, v, \alpha \nvDash p, \gamma$, for any $\gamma \in [X \to M]$.

Corollary 3.2. *Lemma 3.2 implies that for any provable sequent* $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ *if* $\vdash \phi \to false$ *then* $(p_1, \ldots, p_n, p_{n+1})$ *is infeasible.* ■

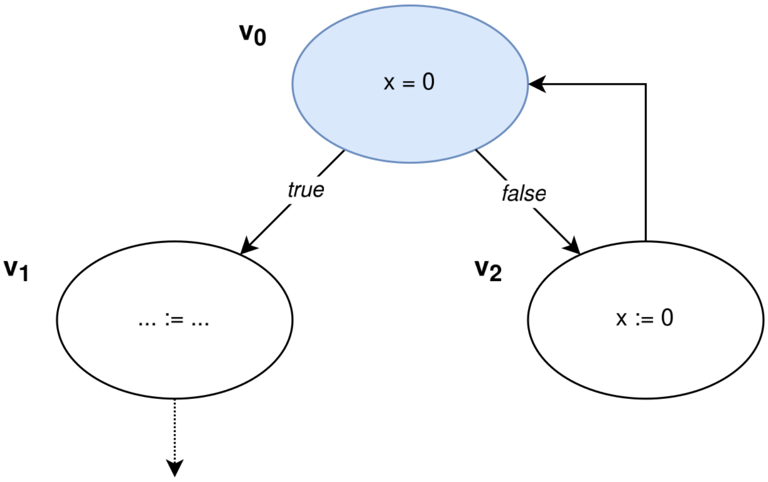


Figure 3.4: Loop impact on the weakest precondition

Proof. By Lemma 3.2 ϕ is necessary, and with Proposition 3.3 if for any $k \in \mathbb{N}$, $\delta^*_G(v_0, \alpha)(k) = (p_1, \beta)$ with $\beta \in [X \to M]$ and $M, \beta \nvDash \phi$ then $p = (p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v_0, \alpha)$ at k. Since $p = (p_1, \ldots, p_n, p_{n+1})$ can only match an infinite sequence of nodes at an index k' if the node at index k' is p_1, if there is no k such that $\delta^*_G(v_0, \alpha)(k) = (p_1, \gamma)$ with $\gamma \in [X \to M]$ and $M, \gamma \vDash \phi$, then p does not match $\delta^*_{G_{node}}(v_0, \alpha)$.

Finally, since $\vdash \phi \to \mathit{false}$ there is no $\alpha \in [X \to M]$ such that $M, \alpha \vDash \phi$ and so

$$G, v_0, \mathit{true} \nvDash (p_1, \ldots, p_n, p_{n+1}), \mathit{true}$$

□

The necessary precondition is not yet a weakest precondition, as there may be loops from $(p_1, \ldots, p_1)$ that change the weakest precondition, as presented in the following example.

Example 3.6.2. Fig. 3.4 presents a concise example of the impact of loops on the weakest precondition of paths. With our precondition calculus, we can obtain $G, v_0, x = 0 \vdash (v_0, v_1), \mathit{true}$. $x = 0$ is indeed a precondition formula for (v_0, v_1), but not the weakest precondition for it. Suppose we execute the graph starting in v_0 with a state $\alpha \in [X \to M]$ such that $\alpha(x) \neq 0$. We will take the edge (v_0, v_2) and in v_2 we update x to be 0. As we now satisfy the necessary precondition $x = 0$ we will take path (v_0, v_1) after exiting v_2. As such the weakest precondition formula for (v_0, v_1), obtained semantically is true. While this loop is trivial there may be much more complex loops that affect the weakest precondition. ■

From the previous example it can be seen that our precondition calculus can achieve the weakest precondition of paths starting in their first node, with some caution on loops. This is the essence of the following theorem.

Theorem 3.3 (Weakest preconditions obtained by the precondition calculus)**.** *If $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ is a provable sequent by the rules of the precondition calculus formed by Eqs. (3.3) to (3.12) without using Eq. (3.12) with $x_1, \ldots, x_m \in Vars(G)$ the free variables of ϕ and there exists no loop $l = (p_1, \ldots, v, \ldots, p_1)$ such that for some $x_i \in \{x_1, \ldots, x_m\}$, $L_V(v) = x_i := t$ and such that $(p_1, \ldots, p_n, p_{n+1})$ is not a sub-path of l, then ϕ is a weakest precondition for $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1.* ■

Proof. Suppose $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ is provable without using Eq. (3.12). Let $v \in V$ a node of G and $\alpha \in [X \to M]$ an assignment. Let $x_1, \ldots, x_m$ the free variables of ϕ.

By Lemma 3.2 ϕ is necessary, and with Proposition 3.3 if for any $k \in \mathbb{N}$, $\delta^*_G(v, \alpha)(k) = (p_1, \beta)$ with $\beta \in [X \to M]$ and $M, \beta \nvDash \phi$ then $p = (p_1, \ldots, p_n, p_{n+1})$ does not match $\delta^*_{G_{node}}(v, \alpha)$ at k. Since $p = (p_1, \ldots, p_n, p_{n+1})$ can only match an infinite sequence of nodes at an index k' if the node at index k' is p_1, if there is no k such that $\delta^*_G(v, \alpha)(k) = (p_1, \gamma)$ with $\gamma \in [X \to M]$ and $M, \gamma \vDash \phi$, then p does not match $\delta^*_{G_{node}}(v, \alpha)$, which in turn implies $\alpha \notin WP_{p,v}$.

Suppose there is no loop $l = (p_1, \ldots, v, \ldots, p_1)$ such that for some $x_i \in \{x_1, \ldots, x_m\}$, $L_V(v) = x_i := t$ and such that $(p_1, \ldots, p_n, p_{n+1})$ is not a sub-path of l. Thus taking any other loop does not change the value of the evaluation of ϕ before and after the loop. This implies that there is no $\alpha \in [X \to M]$ such that $M, \alpha \nvDash \phi$ and for some $k \in \mathbb{N}$, $\delta^*_G(p_1, \alpha) = (p_1, \beta)$ with $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$. Thus $\alpha \notin WP_{p,p_1}$, and only the assignments β such that $M, \beta \vDash \phi$ have the property $\beta \in WP_{p,p_1}$.

Since ϕ is a precondition for p starting in p_1, all $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$ have the property $\beta \in WP_{p,p_1}$. Finally, since we proved above for all $\alpha \in [X \to M]$ such that $M, \alpha \nvDash \phi$ have the property $\alpha \notin WP_{p,p_1}$, WP_{p,p_1} is the set of all $\beta \in [X \to M]$ such that $M, \beta \vDash \phi$ and ϕ is a weakest precondition formula for $p = (p_1, \ldots, p_n, p_{n+1})$ starting in p_1. □

In particular, for an elementary path p where $p_1 = v_0$, we obtain a weakest precondition formula for p if there is no loop from v_0 to v_0 updating the free variables of the precondition formula obtained by our precondition calculus. Thus we obtain the set of all initial assignments for all test cases of p.

Example 3.6.3. In Example 3.5.1 we derive several proof trees. For path $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ we obtain precondition formula $((1 \leq y) \wedge (1 \leq x))$. Since there is no loop $(v_0, \ldots, v_0)$, this is a weakest precondition formula for $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$. By the same argument $(\neg(1 \leq y))$ is a weakest precondition formula for path $(v_0, v_1, v_2, v_3, \tau)$.

However if we derive the proof for path (v_5, v_8, v_9, v_{10}) we obtain precondition formula $(\neg((1 \leq j) \wedge (j \leq x)))$ for (v_5, v_8, v_9, v_{10}) starting in v_5. This is not guaranteed to be a weakest precondition formula since loop (v_5, v_6, v_7, v_5) updates the free variable j of $(\neg((1 \leq j) \wedge (j \leq x)))$. Semantically, we may obtain that the weakest precondition for (v_5, v_8, v_9, v_{10}) starting in v_5 is $[X \to M]$, as taking loop (v_5, v_6, v_7, v_5) a finite number of times will guarantee that the false exit is taken afterwards. ■

Example 3.6.4. In Example 3.5.1 we derive several proof trees. For path $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ we obtain weakest precondition formula $((1 \leq y) \wedge (1 \leq x))$, and for $(v_0, v_1, v_2, v_3, v_4, v_5, v_8)$ we obtain precondition formula $((1 \leq y) \wedge (\neg(1 \leq x)))$ which is also a weakest precondition formula as there is no loop $(v_0, \ldots, v_0)$. It can be remarked that the weakest precondition they represent are distinct from one another (additionally, $\vdash (((1 \leq y) \wedge (1 \leq x)) \wedge ((1 \leq y) \wedge (\neg(1 \leq x)))) \to \mathit{false}$). So $(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7) \not\sim (v_0, v_1, v_2, v_3, v_4, v_5, v_8)$.

Take the following derivation:

$$\dfrac{\dfrac{\dfrac{}{G, v_2, \mathit{true} \vdash (v_2, v_3), \mathit{true}}\ \text{Axiom-Assign}}{G, v_1, \mathit{true} \vdash (v_1, v_2, v_3), \mathit{true}}\ \text{Assign}}{G, v_0, \mathit{true} \vdash (v_0, v_1, v_2, v_3), \mathit{true}}\ \text{Assign}$$

Observe that for any reachable path $p \in \mathit{Path}(G)$ (which implies $WP_{p,v_0} \neq \varnothing$, the proof of which we leave to the reader), since the weakest precondition formula for (v_0, v_1, v_2, v_3) is true then $p \implies (v_0, v_1, v_2, v_3)$ and (v_0, v_1, v_2, v_3) can be safely removed from any test suite that contains any other reachable path. ■

In some simple cases, such as when the program does not contain loops, it is possible to tell whether the precondition formula obtained is a weakest precondition formula or not. However, it is not true in the general case, as there may be an infinite number of paths to check. This includes when a human reader may assert the conditions to obtain a weakest precondition formula are fulfilled.

Theorem 3.4 (Undecidability of weakest precondition formulas)**.** *Let* $G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ *be a provable statement. It is undecidable whether* ϕ *is a weakest precondition calculus or not in the general case.* ■

Proof. By counter-example.

G may be such that there are an infinite number of loops to be verified whether they update the free variables of ϕ or not, thus there is no effective method to decide whether ϕ is a weakest precondition formula or not. □

3.6.3 Achieved completeness

While our precondition calculus is incomplete in general (see Theorem 3.2), it is complete with regard to truths $G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$ when there is no loop $(p_1, \ldots, p_1)$ that updates the free variables of ϕ and that does not contain $(p_1, \ldots, p_n, p_{n+1})$. The proof requires the previous theorems.

Theorem 3.5 (Achieved completeness by the precondition calculus)**.** *The proof system for the precondition calculus formed by rules Eqs.* (3.3) *to* (3.12) *is* complete with regards to truths $G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$ for which we can obtain a weakest precondition formula of $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1*, i.e. for all paths* $(p_1, \ldots, p_n, p_{n+1})$*, all finite formulas* $\phi \in QFFOL(\Sigma, X)$ *with* $x_1, \ldots, x_m \in Vars(G)$ *the free variables of* ϕ *and there exists no loop* $l = (p_1, \ldots, v, \ldots, p_1)$ *such that for some* $x_i \in \{x_1, \ldots, x_m\}$*,* $L_V(v) = x_i := t$ *and such that* $(p_1, \ldots, p_n, p_{n+1})$ *is not a sub-path of* l*, if*

$$G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true$$

then

$$G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$$

is provable. ■

Proof. Suppose $G, p_1, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), true.$ We thus know that ϕ is a precondition formula for $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1. Then by the precondition calculus we can prove a sequent $G, p_1, \chi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ without using the Consequence rule (Eq. (3.12)), where $\chi \in QFFOL(\Sigma, X)$ is a precondition formula for $(p_1, \ldots, p_n, p_{n+1})$ starting in p_1 with free variables $x_1, \ldots, x_m \in Vars(G)$.

Moreover, suppose there exists no loop $l = (p_1, \ldots, v, \ldots, p_1)$ such that for some $x_i \in \{x_1, \ldots, x_m\}$, $L_V(v) = x_i := t$ and $(p_1, \ldots, p_n, p_{n+1})$ is not a sub-path of l, then by Theorem 3.3 χ is a weakest precondition formula for

$(p_1, \ldots, p_n, p_{n+1})$ starting in p_1. Let T be a first-order axiomatic theory formed by the weakest precondition formulas for all paths $(q_1, \ldots, q_l, q_{l+1}) \in Path(G)$ of G starting in q_1 computable by our calculus. Then, since χ is one such weakest precondition formula, $T \vdash \phi \rightarrow \chi$ holds.

Finally, by the Consequence rule (Eq. (3.12)), we obtain

$$G, p_1, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$$

□

3.7 Calculus comparison

In this section we compare our precondition calculus to that of Basu and Yeh [30]. The authors base themselves on Dijkstra's predicate transformer calculus [28], with a form closer to our own calculus. It is to note that the authors observe only programs as complete graphs whereas our approach is based on paths in a graph representation of programs. Furthermore, Basu and Yeh study ”nondeterminable” programs. Understand the usage of the term ”nondeterminable” in [28, 30] as an implication that the program is analysed as a black box whose execution path for a given input state is unknown. This in turn implies that the termination behaviour of the program, i.e. for which input states the program terminates, is unknown. In opposition, our calculus fixes a certain part of the execution path as required of a glass box model. We can furthermore require termination by selection of a path ending in τ. Finally, Basu and Yeh always consider a postcondition predicate Q* whereas our precondition calculus accepts any postcondition, symbolised by the postcondition formula $true$. It is in this sense more general and is fitting for test case generation. The oracle problem to decide whether the output state will satisfy a certain postcondition may be treated with a postcondition calculus.

We now compare the authors' rules to our calculus. Let P, Q be predicates of FOL, S, $S1$ and $S2$ be programs and $WP(S, Q)$ the weakest precondition (as a set of states) of S after the program's execution satisfying predicate Q and guaranteeing termination of S.

(i) *axiom of assignment*: $WP(\text{”}x := E\text{”}, Q) \equiv QE^x$ where QE^x is the predicate obtained by the substitution of the free occurrences of x by E

*This is also the case for Dijkstra's work, but we retain the notations of Basu and Yeh in this paper.

in Q.

This is similar to our Assign rule (Eq. (3.11)) w.r.t. the substitution and is equivalent to Axiom-Assign (Eq. (3.3)) if $Q \equiv true$.

(ii) *axiom of selection*: $WP(\text{"}if\ \ B\ \ then\ \ S1\ \ else\ \ S2\text{"}, Q) \equiv [B \wedge WP(S1, Q)] \vee [\neg B \wedge WP(S2, Q)]$.

The observation of a single branch of a conditional is a major difference between our calculus and those similar to Dijkstra's. While the latter considers all branches at once in the calculus, our calculus only considers one exit. This is because there is no path that can take both exits simultaneously.

By independent analysis of the components of the weakest precondition obtained by Basu and Yeh's axiom of selection, we find similarities with our calculus. $[B \wedge WP(S1, Q)]$ is similar to $(cond \wedge \phi)$ (note $(cond \wedge \phi) \equiv (\phi \wedge cond)$) as obtained by Cond-True (Eq. (3.9)) and equivalent if $Q \equiv true$. It may be noted that in the case that $S1$ is an empty program then Axiom-Cond-True-Exit (Eq. (3.6)) holds. $[\neg B \wedge WP(S2, Q)]$ is similar to $((\neg cond) \wedge \phi)$ as obtained by Cond-False (Eq. (3.10)) and equivalent if $Q \equiv true$. It may be noted that in the case that $S2$ is an empty program then Axiom-Cond-False-Exit (Eq. (3.8)) holds.

(iii) *axiom of composition* $WP(\text{"}S1; S2\text{"}, Q) = WP(S1, WP(S2, Q))$ where ";" is the concatenation operator.

There is no concatenation rule in our calculus. Rather, our conditional rules Cond-True, Cond-False and Assign (Eqs. (3.9) to (3.11)) contain this concatenation as they re-use the precondition formula ϕ computed for the equivalent of $S2$. Furthermore, under the conditions of Theorem 3.3 the theorem guarantees that we obtain the weakest precondition formula by the conditional rules.

(iv) *axiom of iteration* $WP(\text{"}while\ \ B\ \ do\ \ S1\text{"}, Q) \equiv [\neg B \wedge Q] \vee [B \wedge WP(S1, WP(\text{"}while\ B\ do\ S1\text{"}, Q))]$.

Well-structured "while" loops must be modelled in our graph representation as a loop $l_B, l_{S1_1}, \ldots, l_{S1_n}, l_B$ where l_B contains a conditional, $L_E(l_B, l_{S1_1}) = true$ and $l_{S1_1}, \ldots, l_{S1_n}$ is the graph representation of one branch of $S1$ (if $S1$ contains conditionals or other loops, we cannot represent it with a single path). Then by our calculus we can obtain a precondition for $l_{S1_1}, \ldots, l_{S1_n}, l_B$ which we denote ϕ. Then by Cond-True (Eq. (3.9)) we obtain precondition formula $(cond \wedge \phi)$, similar to

$[B \wedge WP(S1, WP(\text{"}while\ B\ do\ S1\text{"}, Q))]$. In the case where $\neg B$ (or $(\neg cond)$ in our calculus) holds, then the case is similar to the axiom of selection's second case with an empty program.

More specifically, the constraints of Theorem 3.3 may not hold, in which case we do not necessarily obtain a weakest precondition formula. We would only obtain a weakest precondition formula if the free variables of $(cond \wedge \phi)$ are not updated in $l_B, l_{S1_1}, \ldots, l_{S1_n}, l_B$, but this implies there is no possibility to take the false exit and terminate, in contradiction with Basu and Yeh's requirement for a weakest precondition. The authors also propose a fixpoint interpretation of the weakest precondition that we do not treat here.

3.8 Summary

Software errors are widely different in their origin and symptoms. In this work we use the division by 0 as a standard error and introduce the different levels that can handle an error. Our model's formalisation begins with the formal definition of a minimal, extendable signature. The extensions to the minimal signature and the corresponding structure are for the reader to provide. From our formal language, we define CFGs to model tested programs. The CFGs are executable, and we introduce the pseudo-node τ to model the termination of the execution. We formally define the notion of coverage, first with assignments, and then with FOL formulas. We then define common testing concepts such as infeasible paths in our approach. The rules of our precondition calculus let us obtain a precondition formula for a given path, and soundness guarantees that the obtained formula covers the path. The proof system is also proved incomplete in the general case. A weakest precondition for a path starting in a given node is the set of assignments that cover the path, starting from the aforementioned node. We define a weakest precondition formula for a path as a FOL formula that is satisfied by only and all the assignments of the weakest precondition for the path. Through the analysis of weakest preconditions, we can reason on the relations between paths, and by extension between test requirements. Notably, when a test requirement p strongly covers a test requirement q, all test cases of p are also test cases of q, and q can be safely removed from the test requirements set. Our precondition calculus guarantees that the precondition formulas it obtains are weakest precondition formulas when there exists no loop that updates the obtained formula's free variables and it does not contain the studied path. It is undecidable in the general case whether these conditions hold or not.

Finally, we present in Table 3.1 a summary of the notation used in this chapter.

Table 3.1: Summary of important notation

Symbol(s)	**Explanation / definition**
Σ	S-sorted first-order signature
M	Many-sorted first-order Σ-structure
$Mod(\Sigma)$	Set of Σ-structures
X	S-indexed family of variable symbols
$[X \to M]$	Set of all assignments
$\alpha[x/a]$	Substitution of variable x by the value a in the assignment $\alpha \in [X \to M]$
$T(\Sigma, X)_s$	Set of terms of sort $s \in S$ over Σ and X
$eval_{M,\alpha}$	Family of evaluation mappings for terms w.r.t. M and $\alpha \in [X \to M]$
$FOL(\Sigma, X)$	Set of First-Order Logic (FOL) formulas over Σ and X
$QFFOL(\Sigma, X)$	Set of quantifier-free FOL formulas over Σ and X
$\phi[x/t]$	Substitution of the free occurrences of variable x by term t in $\phi \in FOL(\Sigma, X)$
$\mathcal{Z}_{div}$	Ordered ring of integers with integer division
$CFG(\Sigma, X)$	Set of well-formed CFGs over Σ and X
G	A well-formed CFG
v_0	Entry node of G
L_V	Mapping of vertices of G to programming statements
L_E	Mapping of edges G to two possible types of exits ($true$ and $false$)
δ_G	State transition function on G
δ_G^*	Small step semantics of G
$\delta_{G_{node}}^*(v, \alpha)$	Execution path of G starting in v with state α ($\delta_{G_{node}}^*(v, \alpha) = \delta_G^*(v, \alpha)_1$)

Continued on next page

Table 3.1: Summary of important notation (Continued)

Symbol(s)	**Explanation / definition**
$\delta^*_{G_{state}}(v, \alpha)$	Execution state sequence of G starting in v with state α ($\delta^*_{G_{node}}(v, \alpha) = \delta^*_G(v, \alpha)_2$)
$Vars(G)$	Set of variables of G
tc	Test case on G
α_0	Initial assignment at the entry of G
$M, \alpha \vDash \phi$	$\alpha \in [X \to M]$ satisfies $\phi \in FOL(\Sigma, X)$ under M
$G, v, \alpha \vDash (p_1, \ldots, p_n, p_{n+1}), \beta$	Coverage relation with assignments
$G, v, \phi \vDash (p_1, \ldots, p_n, p_{n+1}), \chi$	Coverage relation with FOL formulas
$G, v, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), \chi$	Inference sequent
$WP_{p,v}$	Weakest precondition for path p starting in node v
$p \Rightarrow q$	p strongly covers q
$p \to q$	p weakly covers q
$p \nsim q$	p is independent of q

Chapter 4

Discussions and future work

This chapter addresses our work's limitations and future work to improve our model. It also discusses relations between our work and common problems in computer science.

4.1 Limitations

Due to the breadth of the problem, only some of the initial goals have been satisfactorily met. In this section we will focus on some of the remaining issues that should be addressed to improve the model.

4.1.1 Incompleteness

The main limitation of this work is the inability of the calculus to prove sequents of the form $G, v, \phi \vdash (p_1, \ldots, p_n, p_{n+1}), true$ where $v \neq p_1$.

Taking the example of Fig. 3.1, the weakest precondition obtained semantically for the path (v_5, v_8, v_9, v_{10}) starting in v_0 is the set $\{\alpha | \alpha \in [X \to M], \alpha(y) \geq 1\}$. Our precondition calculus cannot obtain a weakest precondition formula for (v_5, v_8, v_9, v_{10}) starting in v_0. We may obtain a precondition formula for (v_5, v_8, v_9, v_{10}) starting in v_5 with our calculus which is $(\neg((1 \leq j) \wedge (j \leq x)))$ as shown by the derivation:

$$\dfrac{\dfrac{\dfrac{}{G, v_9, true \vdash (v_9, v_{10}), true}\ \text{Axiom-Assign}}{G, v_8, true \vdash (v_8, v_9, v_{10}), true}\ \text{Assign}}{G, v_5, (\neg((1 \leq j) \wedge (j \leq x))) \vdash (v_5, v_8, v_9, v_{10}), true}\ \text{Cond-False}$$

Two problems may be observed. The first is the absence of variable y in the formula, as the calculus needs to go through node v_3 to incorporate it into the formula. The second is that the formula contains x whereas the assignments in the weakest precondition for (v_5, v_8, v_9, v_{10}) starting in v_0 are not restricted by the value of x. The reason for the second problem is that the precondition formula is a necessary precondition as defined in the methodology, but not a weakest precondition formula. We may attempt to obtain a weakest precondition formula for path $(v_0, v_1, v_2, v_3, v_4, v_5, v_8, v_9, v_{10})$ starting in v_0, which is $((1 \leq y) \wedge (\neg(1 \leq x)))$. However, with this construction we prevent taking path (v_5, v_6, v_7, v_5), whereas for any $x \geq 1$ and a fixed $y \geq 1$ the (v_5, v_6, v_7, v_5) loop will be taken x times and then path (v_5, v_8, v_9, v_{10}) will be executed. An attempt to prove the prior statement with the precondition calculus would require to obtain the proofs of all sequents $G, v_5, \phi \vdash (v_5, v_6, v_7, \ldots v_5, v_8, v_9, v_{10})$ for all $x \geq 1$ iterations of (v_5, v_6, v_7) in the path, that is we need an infinite number of proofs.

4.1.2 Weakest preconditions

The calculus may obtain a weakest precondition formula corresponding to the semantic weakest precondition without proof that it is indeed a weakest precondition formula. Take for example the trivial path (v_5, v_6, v_7) from Fig. 3.1. The derivation of the proof is:

$$\dfrac{\dfrac{}{G, v_6, \mathit{true} \vdash (v_6, v_7), \mathit{true}}\ \text{Axiom-Assign}}{G, v_5, ((1 \leq j) \wedge (j \leq x)) \vdash (v_5, v_6, v_7), \mathit{true}}\ \text{Cond-True}$$

The obtained precondition formula $((1 \leq j) \wedge (j \leq x))$ for (v_5, v_6, v_7) starting in v_5 corresponds to the correct semantically-obtained weakest precondition. But since there exists the loop $(v_5, v_8, v_9, v_{10}, v_3, v_4, v_5)$ that modifies the free variable j in v_4 we are not guaranteed to obtain a weakest precondition formula for (v_5, v_6, v_7) starting in v_5. Since v_4 is on all paths from v_0 to v_5, it is possible to circumvent the issue by extending the proof to the path (v_4, v_5, v_6, v_7) for which the proof derivation is:

$$\dfrac{\dfrac{\dfrac{}{G, v_6, \mathit{true} \vdash (v_6, v_7), \mathit{true}}\ \text{Axiom-Assign}}{G, v_5, ((1 \leq j) \wedge (j \leq x)) \vdash (v_5, v_6, v_7), \mathit{true}}\ \text{Cond-True}}{G, v_4, ((1 \leq 1) \wedge (1 \leq x)) \vdash (v_4, v_5, v_6, v_7), \mathit{true}}\ \text{Assign}$$

Here, since the free variable j is not present any more in the obtained precondition formula and there is no loop of G that updates x, the precondition formula for (v_4, v_5, v_6, v_7) starting in v_4, is guaranteed to be a weakest precondition formula for (v_4, v_5, v_6, v_7) starting in v_4.

However some issues cannot be circumvented this way. The example of path (v_5, v_8, v_9, v_{10}) above shows that our calculus can't obtain a weakest precondition formula for this path.

One last limitation of the model with regard to weakest preconditions is the case of programs whose graph models contain loops of the form $(v_0, \dots, v_0)$. In such case it is probable that the conditions to obtain a weakest precondition formula are rarely met. One such program can be a server which continuously listens to incoming requests and replies to them on loop.

4.1.3 Error handling

In the methodology, we mentioned the inadequacy of the logic to model error handling at the language's runtime and operating system levels without modifications to the calculus and the modelling of the graph. We conjecture such modifications may include:

(i) Meta-jumps between different graphs: for example the try-catch block of a (Java-like) program P may be a graph on its own, and termination is replaced by a jump to the graph of P, whether the termination is normal or error-based. Similarly, meta-jumps could be used to access the operating system's graph, at the cost of a large complexity addition.

(ii) Additional exits: instead of meta-jumps, one other possibility is to allow more exits for each type of node, depending on the context. To reuse the try-catch example, all nodes in the try block could have an additional exit to the entry of the catch block. Similarly the runtime or operating system could be modelled as a disjoint set of nodes to that of the program, and exits to and from this disjoint set let us model the interactions between the runtime or operating system and the program.

We also noted in the methodology that error handling at the data type level may be achieved through specific error variables. This may be achieved by updating the rules of inference alone, however the exact design (e.g. one error variable for all kinds of errors, one per sort, etc.) will substantially change the updates to be performed, and as such we do not explore this point further.

4.2 Future work

4.2.1 Postcondition calculus

Our work introduces a precondition calculus yet lacks a postcondition calculus to automatically generate possible test oracles. We have considered the possibility of modifying the precondition calculus to incorporate any postcondition instead of $true$ similarly to the work of Dijkstra [28], but this approach would not meet the needs of testers. The precondition calculus computes the precondition by executing the graph backwards, in contrast a tester would want a forward execution and by giving a precondition would want to obtain a postcondition. As such we consider the best approach is to compute a precondition ϕ for a certain path with the precondition calculus, and take a stronger precondition χ such that for some axiomatic theory of the data types T, there is that $T \vdash \chi \to \phi$. Then the postcondition calculus would let us obtain a strongest postcondition formula for the path, starting with precondition formula χ.

The main problem with forward execution of a graph is the updates of variables, including updates based on their previous value, to a (syntactic) formula. For example, suppose we obtain postcondition $x \leq 1$ for a path p and we next treat a node v such that $L_V(v) = x := x{+}1$. The resulting postcondition should be $x \leq 1 + 1$, or equivalent formulas such as $x - 1 \leq 1$. However we cannot find a general modification of the formula that captures this behaviour. To our knowledge the literature presents only syntactic effects for backward execution (see [28, 30]) and forward effects are presented semantically, such as in Floyd [38]. At first the application of the update to the other terms in the relation was envisioned, e.g. for $x \leq c$ with c an integer constant then after treating node v with $L_V(v) = x := x + 1$, the postcondition would become $x \leq c + 1$. Now suppose a more complex relation such as $I \in \Sigma_{int\ string}$[*] whose solutions are the set of integers x and strings s such that the character of index x in s is the character a. Suppose we have postcondition $I(x, s)$ for a certain path and we go through node v with $L_V(v) = x := x{+}1$. The envisioned update to the other terms would lead to the new postcondition $I(x, s + 1)$, but the $+ \in \Sigma_{int\ int,int}$ operation is not defined for strings and integers. Another envisioned method was to replace the variable by the term where all operations are replaced by their inverse. Employing again the postcondition $x \leq 1$ for a path p and node v with $L_V(v) = x := x + 1$, the resulting postcondition after

[*] We do not define the $string$ sort in this example, but assume it is an indexed list of characters.

treating v would be $x - 1 \leq 1$. The limitation to this approach is that an operation may not have an inverse (e.g. for a surjective operation), in which case the postcondition calculus could not be constructed.

A last issue with regard to postcondition calculi to be addressed is whether to add constraints to variables when they are assigned a constant if they are not free variables of the precondition. For example the program of Fig. 3.1 returns variable $result$, whose value after the execution of the program and its graph is relevant for testers. For example any execution of $(v_0, v_1, v_2, v_3, \tau)$ of Fig. 3.1 should guarantee that $result$ is 0 after the execution of the program. No precondition obtained by our precondition calculus contains the $result$ variable as it is not part of any conditional node's formula. It is possible to use a stricter precondition that has some variables such as $result$ as free variables, but we conjecture this approach will not be precise enough to be formalised. For example if we have precondition formula $(\phi \wedge (result \leq -1))$ for $(v_0, v_1, v_2, v_3, \tau)$ starting at v_0 such that $result$ is not a free variable of ϕ, should the postcondition after executing v_0 that maps $result$ to 0 be $(\phi \wedge (0 \leq -1))$? Or $(\phi \wedge (result \leq 0))$? We instead propose that the postcondition calculus should overwrite the atomic formulas concerning variable x*, resulting in a formula χ and use formula $(\chi \wedge (x = c))$ as the updated postcondition formula, when a node v with $L_V(v) = x := c$ is treated, with c a constant symbol.

4.2.2 Stricter structure flow graphs

Our model is close to that of flowcharts†, with loosely structured control flows. Notably, our work allows to model goto-like control flows. Many modern programming languages only allow structured control flows and either restrict or ban the use of goto statements. A possible improvement for our work is thus to restrict the control flows that a well-formed CFG should allow. For example we conjecture that while loops should be defined as loops in the graph such that they contain at least one conditional node v, termed the loop entry, and all loops $(v, v', \dots, v)$ are such that $L_E(v, v') = true$ and there is no path $(v_0, \dots, v_l)$ with v_l a node of the while loop that does not contain the entry node v. Similarly, all paths from v_l to any node that is not in the subgraph of the while loop must contain the edge (v, v'') with $L_E(v, v'') = false$.

*Which would require a formal definition that handles how to process the formulas of the form $(r(x, \dots) \wedge \chi)$, $(\neg(r(x, \dots)))$, etc.

†We let the reader refer to Loeckx and Sieber [39] for a reference w.r.t. flowcharts.

4.2.3 Automation of test case generation

Due to time constraints we have not been able to automate the precondition calculus for test case generation. However this can be done given elementary paths by first automating the derivation of a proof tree to obtain a precondition formula for each elementary path, and then obtaining the assignments that satisfy each precondition formula. Automating the derivation is possible through a custom-made algorithm matching the premises and side-conditions of our inference rules (the first inference is always an axiom and no other sequent inferences should be axioms of our calculus). Once the precondition is obtained, a theorem prover such as Z3 [40] can be used to obtain the assignments that satisfy the path's precondition formula. A tester only needs to select one such assignment to obtain a test case for the path.

4.3 Relation to other problems

Our work can model termination through the pseudo-node τ. This opens a relation to the halting problem. We may alter a program so that all exit nodes point towards a new node v_{exit} and then the halting problem is equivalent to obtaining $WP_{(v_{exit},\tau),v_0}$ (note that we can only consider paths of at least two nodes with our calculus). It is well-known that the halting problem is undecidable in the general case. This implies that for any CFG, no calculus (ours included) can determine a weakest precondition formula for (v_{exit}, τ).

In the limitations we highlight the issue of loops that prevent us to obtain a weakest precondition formula. Loops have invariants, formulas that hold before and after each iteration of a loop. In the case of loop (v_5, v_6, v_7, v_5) from Fig. 3.1, the reader may ascertain that the value of x is not updated by the loop, so $x = c$ with c an integer constant is an invariant for the loop. Since the loop increments the value of j, at some point j's value will be superior to x's value, and we exit the loop to node v_8. Thus it can be semantically ascertained that path (v_5, v_6, v_7, v_5) will be taken a finite number of times, and path (v_5, v_8) will always follow. Thus loop (v_5, v_6, v_7, v_5) (and path (v_5, v_6, v_7) as the only exit of v_7 is to v_5) strongly covers path (v_5, v_8), although the model being syntactic cannot ascertain it. However, it is undecidable whether a formula is a loop invariant in the general case. This implies that no calculus for any signature, structure and graph can rely entirely on invariants to solve the weakest precondition limitations. We let the reader refer to the work of Kovács and Varonka [41] for more properties of loops.

Our choice of allowing a single statement per node leads to large graphs.

This could lead to path explosion, a phenomenon where the number of control flows grows exponentially with the size of a program, and so the size of its graph. However, the most commonly used glass-box coverage models (e.g. NC, EC) study short paths, with few control flows. Thus, path explosion is not a concern for our model.

4.4 Summary

Our work is not without limitations. The main issue is the inability of our calculus to prove a sequent when we start in a node v, different from the first node of the studied path p_1. It is also possible that we do obtain a weakest precondition formula but without the guarantee it is one. This can happen when the conditions to obtain a weakest precondition formula are not met. We also discuss how to adapt our model to handle errors at the language's runtime and operating system levels. We present several leads of future work to improve our model. First, we highlight the challenges of a postcondition calculus, notably the syntactic impacts of forward execution and whether to add constraints on a variable when it is assigned. Then we discuss stricter structures for our graphs, followed by the means to automate the logic for test case generation. Finally, we relate our work to common problems of computer science: the halting problem, loop invariants and path explosion.

Chapter 5

Conclusion

5.1 Conclusions

In this paper we have defined and studied a finitary formal model to reason on programs as well-formed CFGs to reason on test requirement redundancy, which is proven sound. The model can obtain a weakest precondition formula for paths under certain conditions on loops, in which case we can reason on test requirement redundancy. The precondition calculus has been proven incomplete in the general case, and is also proved undecidable. However it is complete w.r.t. preconditions for paths $(p_1, \dots, p_n, p_{n+1})$ starting in p_1 when there is no loop $(p_1, \dots, p_1)$ of the graph G that does not contain $(p_1, \dots, p_n, p_{n+1})$ and updates the free variables of the obtained precondition. Due to lack of time we have not been able to automate the model for test case generation but we present the steps to perform the automation in the future work. In turn, automation can lead to the adoption of more powerful coverage models for test suites and the program is thus more reliable.

The model can be extended as we have defined, to use any new sorts and adequate structures that the reader must provide. The precondition calculus' properties are proved for all signatures and structures that extend the minimal signature of the ordered ring of integers with integer division $\mathcal{Z}_{div}$.

Software errors can be captured at the data type level with our model, notably by non-standard values. We present in the limitations conjectures for the modifications of the model to capture software errors at the language runtime and operating system levels. We also present in the future work the challenges of a postcondition calculus to solve the oracle problem. We conjecture there are properties and improvements yet to be found on the model we have developed, as presented in the future work section.

The project was more challenging than expected, despite the minimal signature studied that only includes integers and first-order formulas. This is in part due to the complexity of programs and the underlying mathematical concepts and properties, such as the undecidability of loop invariants as discussed in the limitations. It is our hope that the extensively formal definitions presented in this work compared to the literature in software testing allow other projects to envision formal models of testing and save them time for more testing-relevant problems.

5.2 Ethics and sustainability

Ethically, testing is a necessary quality assurance process as it encompasses the stack of the program. With the large amount of personal data processed and stored in companies' servers, testing can reduce the risk of a security vulnerability and the loss of said personal data. Risks incurred by software do not only affect data, but also the physical well-being of users, notably in the case of programs for medical equipment, where a particularly stringent quality process including extensive testing must take place. Finally, testing can also reduce the risks for the psychological well-being of users, as may be understood from the example of bots that delete adult or inappropriate content from a forum that can be accessed by children. Glass box testing, as it tests the structure of a given program, can expose inappropriate behaviours of the code under some inputs. For example, suppose a program for banks that computes the risk for a person not to reimburse a loan they wish to obtain. The program should not have a different behaviour (and so take different paths or have a different postcondition) for different persons who share the same revenue and other relevant parameters but have different skin colours, religions, political opinions, etc. Although inputs are under the subfield of black-box testing, glass box testing cannot entirely abstract them away as shown by the study of weakest preconditions by our calculus. As such glass box testing can also unravel unethical inputs, with for example the skin colour, religion and political opinion parameters of the above example. Our work has no direct ethical impact that is not inherited from software testing and glass box testing as mentioned above. We hope a more formal approach to glass box testing can improve the quality of the software under test and prevents the hurdles presented in this section.

With regard to sustainability, the thesis does not contribute directly to any Sustainable Development Goals (SDGs) of the United Nations (UN). Instead, in the midst of the digitalisation trend of the world's systems, the improvements to testing by this work can impact the categories of sustainable development in the following way:

- Economy: the cost of lack of testing is a major concern for information technology companies, as the sums reach billions of dollars [1]. Automated test case generation and test requirement redundancy identification can limit the cost of testing, or allow more extensive testing for the same budget.

- Social impact: improved testing can lead to fewer risks for users, whether risks for their data or risks for their well-being induced by the software product. Errors in software for medical uses can notably be fatal to the user.

- Ecology: the execution of test cases consumes resources, including hardware to test on and energy to run the tests. This work can reduce the number of test cases for a given set of test requirements and thus the resources spent on testing. In opposition, automated test case generation may cause an increase of the size of the test suite and thus an increase of the energy spent to execute it. Caution must be advised on this latter point for ecological impacts.

References

[1] S. Planning, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology*, vol. 1, no. 2002, 2002. [Online]. Available: https://www.nist.gov/system/files/documents/2021/03/24/econImpactSumm.v23.pdf [Pages 1 and 81.]

[2] H. Krasner, “The cost of poor software quality in the us: A 2020 report,” *Consortium For Information and Software Quality*, 2021. [Online]. Available: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf [Page 1.]

[3] B. fung, “We finally know what caused the global tech outage - and how much it cost,” *CNN*, 2024. [Online]. Available: https://edition.cnn.com/2024/07/24/tech/crowdstrike-outage-cost-cause/index.html [Page 1.]

[4] H. Ehrig and B. Mahr, “Fundamentals of algebraic specification 1: Equations and initial semantics,” in *Fundamentals of Algebraic Specification*, 1985. [Online]. Available: https://api.semanticscholar.org/CorpusID:59866408 [Pages 2 and 9.]

[5] F. A. Authority, “Do-178b/c differences tool,” 3 2014. [Online]. Available: https://www.faa.gov/sites/faa.gov/files/aircraft/air_cert/design_approvals/air_software/differences_tool.pdf [Pages 5 and 13.]

[6] F. E. Allen, “Control flow analysis,” in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970. doi: 10.1145/800028.808479. ISBN 9781450373869 p. 1–19. [Online]. Available: https://doi.org/10.1145/800028.808479 [Page 7.]

[7] P. Ammann and J. Offutt, *Introduction to software testing*, 2nd ed. Cambridge, United Kingdom ;: Cambridge University Press, 2017. ISBN 978-1-107-17201-2 [Pages 7, 8, 12, 13, 14, and 35.]

[8] A. Wasilewska, *Correction to: Logics for Computer Science*. Cham: Springer International Publishing, 2018, pp. C1–C1. ISBN 978-3-319-92591-2. [Online]. Available: https://doi.org/10.1007/978-3-319-92591-2_12 [Page 11.]

[9] G. Gentzen, "Untersuchungen über das logische schließen. i," *Mathematische Zeitschrift*, vol. 39, no. 1, pp. 176–210, Dec 1935. doi: 10.1007/BF01201353. [Online]. Available: https://doi.org/10.1007/BF01201353 [Page 11.]

[10] ——, "Untersuchungen über das logische schließen. ii," *Mathematische Zeitschrift*, vol. 39, no. 1, pp. 405–431, Dec 1935. doi: 10.1007/BF01201363. [Online]. Available: https://doi.org/10.1007/BF01201363 [Page 11.]

[11] L. Henkin, "The completeness of the first-order functional calculus," *The Journal of Symbolic Logic*, vol. 14, no. 3, pp. 159–166, 1949. [Online]. Available: http://www.jstor.org/stable/2267044 [Page 12.]

[12] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i," *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, 12 1931. doi: 10.1007/BF01700692. [Online]. Available: https://doi.org/10.1007/BF01700692 [Page 12.]

[13] T. B. Dao and E. Shibayama, "Coverage criteria for automatic security testing of web applications," in *Information Systems Security*, S. Jha and A. Mathuria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17714-9 pp. 111–124. [Page 13.]

[14] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9. New York, NY, USA: Association for Computing Machinery, 2001. doi: 10.1145/503209.503244. ISBN 1581133901 p. 256–267. [Online]. Available: https://doi.org/10.1145/503209.503244 [Page 13.]

[15] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans, "Coverage criteria for behavioural testing of software product lines," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*,

T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN 978-3-662-45234-9 pp. 336–350. [Page 13.]

[16] C. R. Prause, J. Werner, K. Hornig, S. Bosecker, and M. Kuhrmann, "Is 100% test coverage a reasonable requirement? lessons learned from a space software project," in *Product-Focused Software Process Improvement*, M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, Eds. Cham: Springer International Publishing, 2017. ISBN 978-3-319-69926-4 pp. 351–367. [Page 13.]

[17] Z. Chen, B. Xu, X. Zhang, and C. Nie, "A novel approach for test suite reduction based on requirement relation contraction," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 390–394. [Pages 14 and 20.]

[18] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, p. 270–285, 7 1993. doi: 10.1145/152388.152391. [Online]. Available: https://doi.org/10.1145/152388.152391 [Page 15.]

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. ISBN 0716710447 [Page 15.]

[20] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *Fundamental Approaches to Software Engineering*, M. B. Dwyer and A. Lopes, Eds. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-71289-3 pp. 291–305. [Page 16.]

[21] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, p. 146–162, oct 1999. doi: 10.1145/318774.318939. [Online]. Available: https://doi.org/10.1145/318774.318939 [Page 16.]

[22] J. Jones and M. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003. doi: 10.1109/TSE.2003.1183927 [Page 16.]

[23] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test

suites," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998. doi: 10.1109/ICSM.1998.738487 pp. 34–43. [Page 16.]

[24] M. Heimdahl and D. George, "Test-suite reduction for model based tests: effects on test quality and implications for testing," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, 2004. doi: 10.1109/ASE.2004.1342735 pp. 176–185. [Page 16.]

[25] Z. Chen, X. Zhang, and B. Xu, "A degraded ilp approach for test suite reduction." in *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, 01 2008, pp. 494–499. [Page 16.]

[26] A. Bajaj and O. P. Sangwan, "Discrete and combinatorial gravitational search algorithms for test case prioritization and minimization," *International Journal of Information Technology*, vol. 13, no. 2, pp. 817–823, 4 2021. doi: 10.1007/s41870-021-00628-8. [Online]. Available: https://doi.org/10.1007/s41870-021-00628-8 [Page 17.]

[27] I. Hooda and R. Chhillar, "Test case optimization and redundancy reduction using ga and neural networks," *International Journal of Electrical and Computer Engineering*, vol. 8, no. 6, p. 5449, 2018. [Page 17.]

[28] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, p. 453–457, aug 1975. doi: 10.1145/360933.360975. [Online]. Available: https://doi.org/10.1145/360933.360975 [Pages 17, 18, 19, 65, and 74.]

[29] C. A. R. Hoare, "Some properties of predicate transformers," *Journal of the ACM*, vol. 25, no. 3, pp. 461–480, 1978. [Page 17.]

[30] S. K. Basu and R. T. Yeh, "Strong verification of programs," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 339–346, 1975. [Online]. Available: https://api.semanticscholar.org/CorpusID:1855419 [Pages 17, 18, 65, and 74.]

[31] W. Ahrendt, C. Gladisch, and M. Herda, *Proof-based Test Case Generation*. Cham: Springer International Publishing, 2016, pp. 415–451. ISBN 978-3-319-49812-6. [Online]. Available: https://doi.org/10.1007/978-3-319-49812-6_12 [Page 18.]

[32] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011. doi: 10.1145/1985793.1985995. ISBN 9781450304450 p. 1066–1071. [Online]. Available: https://doi.org/10.1145/1985793.1985995 [Page 19.]

[33] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: Association for Computing Machinery, 1994. doi: 10.1145/186258.186523. ISBN 0897916832 p. 80–94. [Online]. Available: https://doi.org/10.1145/186258.186523 [Page 19.]

[34] C. Gladisch, "Verification-based test case generation for full feasible branch coverage," in *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, 2008. doi: 10.1109/SEFM.2008.22 pp. 159–168. [Page 19.]

[35] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229 [Page 29.]

[36] *signal(7) — Linux manual page*, 6 2024. [Online]. Available: https://www.man7.org/linux/man-pages/man7/signal.7.html [Page 29.]

[37] K. Meinke, "DD2459 software reliability 2023 take-home exam," 3 2023. [Page 31.]

[38] R. W. Floyd, "Assigning meaning to programs," in *Proceedings of Symposium on Applied Mathematics*, J. T. Swhartz, Ed., vol. 19. American Mathematical Society, 1967. ISBN 978-0-8218-6728-0 pp. 19–32. [Online]. Available: https://api.semanticscholar.org/CorpusID:62710333 [Page 74.]

[39] L. Jacques and S. Kurt, *The Foundations of Program Verification*, ser. Series in Computer Science. Teubner, 1984. ISBN 3-519-02101-3 [Page 75.]

[40] M. Research, “Z3prover,” https://github.com/Z3Prover/z3, 2007. [Page 76.]

[41] L. Kovács and A. Varonka, “What else is undecidable about loops?” in *Relational and Algebraic Methods in Computer Science*, R. Glück, L. Santocanale, and M. Winter, Eds. Cham: Springer International Publishing, 2023. ISBN 978-3-031-28083-2 pp. 176–193. [Page 76.]

TRITA-EECS-EX-2025:58
Stockholm, Sweden 2025

www.kth.se