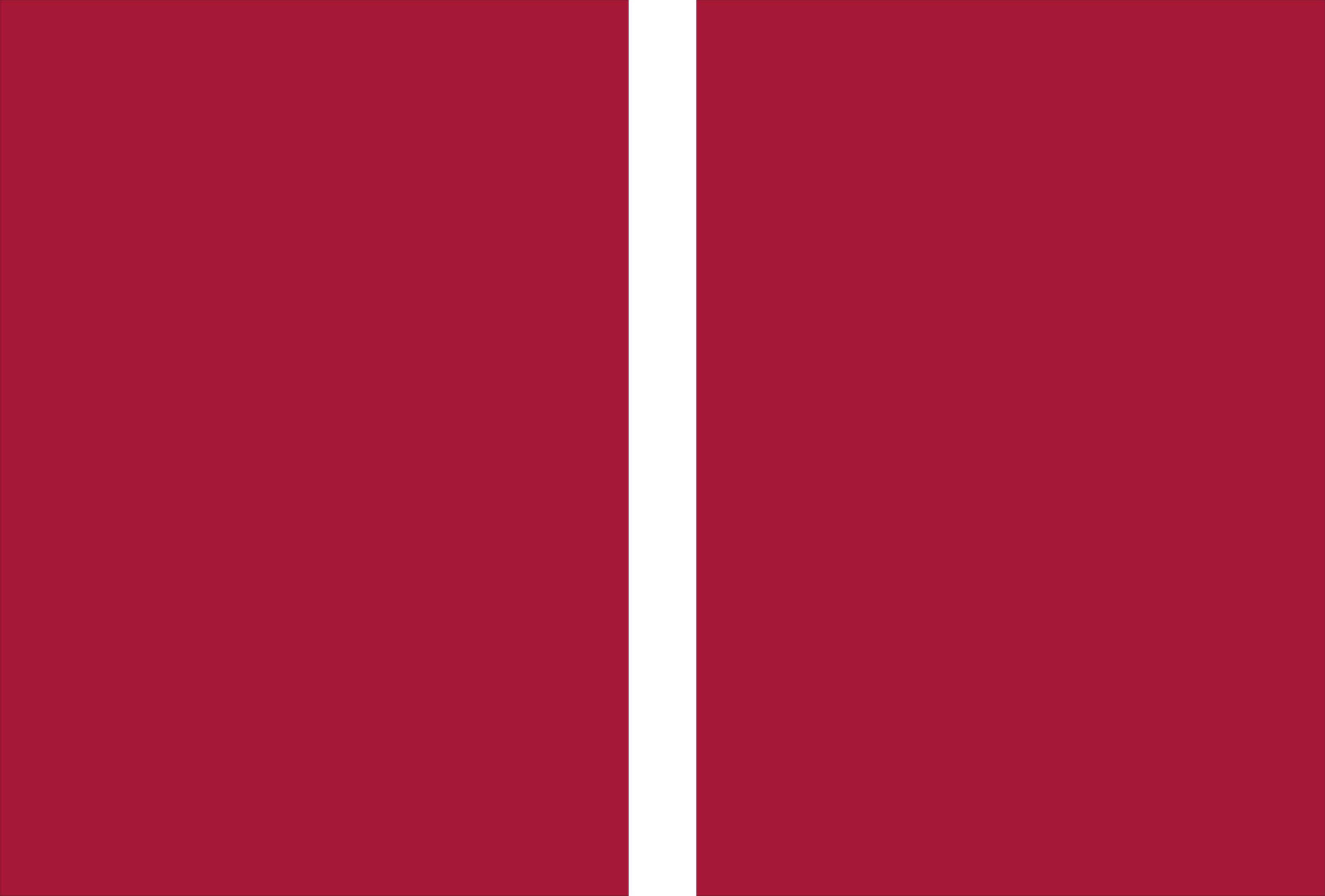


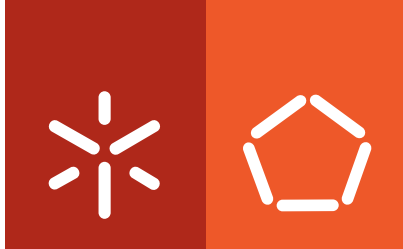


Universidade do Minho
Escola de Engenharia

Daniela Carneiro da Cruz

Verification, Slicing, and Visualization
of Programs with Contracts





Universidade do Minho
Escola de Engenharia

Daniela Carneiro da Cruz

Verification, Slicing, and Visualization of Programs with Contracts

Tese de Doutoramento em Informática,
Especialidade de Ciências da Computação

Trabalho efectuado sob a orientação do
Doutor Pedro Rangel Henriques
e do
Doutor Jorge Sousa Pinto

Junho de 2011

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE APENAS PARA EFEITOS
DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE
COMPROMETE;

Universidade do Minho, ____/____/____

Assinatura: _____

Acknowledgements

There are no words to match my gratitude,
Neither much like Shakespeare I might write,
But, at least in Portuguese I can express myself better than him,
Thus the following acknowledgements are in my mother tongue.

Agradeço ao Professor Pedro Rangel Henriques por desde início me permitir seguir as minhas preferências de investigação, ser uma presença permanente ao longo deste percurso e pelo seu empenho. Mais uma vez, recordo o seu lema que continua presente desde que trabalhamos juntos: “Enquanto houver estrada vamos caminhar, enquanto houver ventos e marés não vamos parar”.

Agradeço ao Professor Jorge Sousa Pinto, pela sua presença, também constante, e pelo seu esforço e dedicação a este projecto. Agradeço-lhe principalmente pela confiança que sempre mostrou em mim e no meu trabalho e pela calma que sempre me transmitiu.

A ambos agradeço as nossas viagens, que foram mais uma exploração do mundo. Cada um com a sua visão e o seu conhecimento, foi um aprendizado que nunca esquecerei.

A ambos agradeço também o facto de serem, mais do que orientadores, verdadeiros amigos!

Agradeço ainda ao Professor José Bernardo Barros pelo contributo que deu para este projecto.

Não poderia deixar de mencionar o grupo de Especificação e Processamento de Linguagens (gEPL) como um todo por sempre estarem dispostos a uma jogatina para aliviar o stress. Queria também agradecer aos meus alunos Daniela, Sérgio, Hugo, Márcio, Zé Luís e Miguel.

Agradeço ao Nuno Oliveira pelo seu apoio permanente e por ser um verdadeiro amigo. Um muito obrigada também à Maria João pela sua amizade e pelo projeto Alma que desencadeou todo este meu percurso académico. Obrigada Necas e Zinha. Obrigada Trengos!

Agradeço à Sofia e ao Bruno pelos momentos de descontração que sempre partilhamos e do incentivo que sempre me deram!

Um agradecimento especial dedico ao Jorge: obrigada por fazeres parte da minha vida! És muito importante para mim.

Por fim, agradeço aos meus pais, minha irmã e especialmente ao meu irmão pelo seu apoio incondicional. Acima de tudo, muito obrigada por me terem ensinado o verdadeiro valor da vida e as coisas que nela realmente tem significado.

Esta tese foi suportada pela Fundação para a Ciência e a Tecnologia (FCT) através da Bolsa de Doutoramento SFRH/BD/33231/2007 e também pelos projectos RESCUE (contrato FCT sob a referência PTDC / EIA / 65862 / 2006) e CROSS (contrato FCT sob a referência PTDC / EIACCO / 108995 / 2008).

E porque não há vida sem poesia:

*Agradeço a quem me ensina,
desde quando pequenina.
Me ensina que a bondade,
é como identidade.
Me ensina que a humildade,
é suporte da igualdade.
Me ensina que a vontade,
já é conseguir a metade.*

Abstract

As a specification carries out relevant information concerning the behavior of a program, why not explore this fact to slice a program in a semantic sense aiming at optimizing it or easing its verification? It was this idea that Comuzzi, in 1996, introduced with the notion of *postcondition-based slicing* — slice a program using the information contained in the postcondition (the condition Q that is guaranteed to hold at the exit of a program). After him, several advances were made and different extensions were proposed, bridging the two areas of Program Verification and Program Slicing: specifically *precondition-based slicing* and *specification-based slicing*. The work reported in this Ph.D. dissertation explores further relations between these two areas aiming at discovering mutual benefits.

A deep study of specification-based slicing has shown that the original algorithm is not efficient and does not produce minimal slices. In this dissertation, traditional specification-based slicing algorithms are revisited and improved (their formalization is proposed under the name of *assertion-based slicing*), in a new framework that is appropriate for reasoning about imperative programs annotated with contracts and *loop invariants*.

In the same theoretical framework, the semantic slicing algorithms are extended to work at the program level through a new concept called *contract-based slicing*. Contract-based slicing, constituting another contribution of this work, allows for the study of a program at an interprocedural level, enabling optimizations in the context of code reuse.

Motivated by the lack of tools to prove that the proposed algorithms work in practice, a tool (**GamaSlicer**) was also developed. It implements all the existing semantic slicing algorithms, in addition to the ones introduced in this dissertation. This third contribution is based on generic *graph visualization and animation* algorithms that were adapted to work with *verification* and *slice graphs*, two specific cases of *labeled control flow graphs*.

Resumo

Tendo em conta que uma especificação contém informação relevante no que diz respeito ao comportamento de um programa, faz sentido explorar este facto para o cortar em fatias (*slice*) com o objectivo de o otimizar ou de facilitar a sua verificação. Foi precisamente esta ideia que Comuzzi introduziu, em 1996, apresentando o conceito de *postcondition-based slicing* que consiste em cortar um programa usando a informação contida na pós-condição (a condição Q que se assegura ser verdadeira no final da execução do programa). Depois da introdução deste conceito, vários avanços foram feitos e diferentes extensões foram propostas, aproximando desta forma duas áreas que até então pareciam desligadas: *Program Verification* e *Program Slicing*. Entre estes conceitos interessa-nos destacar as noções de *precondition-based slicing* e *specification-based slicing*, que serão revisitadas neste trabalho. Um estudo aprofundado do conceito de *specification-based slicing* relevou que o algoritmo original não é eficiente e não produz slices mínimos.

O trabalho reportado nesta dissertação de doutoramento explora a ideia de tornar mais próximas essas duas áreas visando obter benefícios mútuos. Assim, estabelecendo uma nova base teórica matemática, os algoritmos originais de *specification-based slicing* são revistos e aperfeiçoados — a sua formalização é proposta com o nome de *assertion-based slicing*.

Ainda sobre a mesma base teórica, os algoritmos de slicing são estendidos, de forma a funcionarem ao nível do programa; além disso introduz-se um novo conceito: *contract-based slicing*. Este conceito, *contract-based slicing*, sendo mais um dos contributos do trabalho aqui descrito, possibilita o estudo de um program ao nível externo de um procedimento, permitindo, por um lado, optimizações no contexto do seu uso, e por outro, a sua reutilização segura.

Devido à falta de ferramentas que provem que os algoritmos propostos de facto funcionam na prática, foi desenvolvida uma, com o nome **GamaSlicer**, que implementa todos os algoritmos existentes de slicing semântico e os novos propostos. Uma terceira contribuição é baseada nos algoritmos genéricos de *visualização e animação de grafos* que foram adaptados para funcionar com os grafos de controlo de fluxo *etiquetados* e os grafos de *verificação e slicing*.

Contents

1	Introduction	1
1.1	Contributions and Document Structure	3
2	State-of-the-Art: Code Analysis	7
2.1	Basic Concepts	8
2.2	Anatomy of Code Analysis	15
2.2.1	Data Extraction	15
2.2.2	Information Representation	16
2.2.3	Knowledge Exploration	17
2.3	Current Code Analysis Challenges	20
2.3.1	Language Issues	20
2.3.2	Multi-Language Analysis	21
2.3.3	Static, Dynamic and Real-Time Analysis	24
2.3.4	Analyzing Executables	25
2.3.5	Information Retrieval	28
2.3.6	Data Mining	28
2.4	Applications	29
2.4.1	Debugging	29
2.4.2	Reverse Engineering	30
2.4.3	Program Comprehension	30
2.5	Tools	31
2.5.1	FxCop	32
2.5.2	Lint	32
2.5.3	CodeSonar and CodeSurfer	33
3	State-of-the-Art: Program Verification	35
3.1	Basic Concepts	40
3.1.1	Propositional Logic	41
3.1.2	First-order Logic	48
3.1.3	Hoare Logic	57
3.1.4	Other Definitions	62
3.2	Static Techniques: Semi-automated	67
3.2.1	Mechanizing Hoare Logic	70

3.2.2	The Weakest Precondition Strategy	70
3.2.3	A VCGen Algorithm based on Weakest Preconditions	73
3.2.4	The Strongest Postcondition Strategy	75
3.3	Static Techniques: Fully-Automated	76
3.3.1	Abstract Interpretation	76
3.3.2	Model Checking	78
3.4	Dynamic Techniques	84
3.4.1	Runtime Verification	84
3.4.2	Testing	85
3.4.3	Testing Methods	86
3.4.4	Testing Levels	88
3.5	Tools	92
3.5.1	Program Verifiers	93
3.5.2	Static Analysis Tools	98
3.5.3	Model Checking Tools	100
4	State-of-the-Art: Slicing	105
4.1	The Concept of Program Slicing	106
4.1.1	Program Example	106
4.1.2	Static Slicing	107
4.1.3	Dynamic Slicing	108
4.1.4	Quasi-static Slicing	110
4.1.5	Conditioned Slicing	112
4.1.6	Simultaneous Dynamic Slicing	113
4.1.7	Union Slicing	114
4.1.8	Other Concepts	116
4.1.9	Dicing	116
4.1.10	Chopping	117
4.1.11	Relationships among Program Slicing Models	117
4.1.12	Methods for Program Slicing	118
4.2	Static Slicing	119
4.2.1	Basic Slicing Algorithms	119
4.2.2	Slicing Programs with Arbitrary Control Flow	121
4.2.3	Interprocedural Slicing Methods	123
4.2.4	Slicing in the Presence of Composite Datatypes and Pointers	129
4.3	Dynamic Slicing	130
4.3.1	Basic Algorithms for Dynamic Slicing	131
4.3.2	Slicing Programs with Arbitrary Control Flow	135
4.3.3	Interprocedural Slicing Methods	135
4.3.4	Slicing in the Presence of Composite Datatypes and Pointers	136
4.4	Applications of Program Slicing	137
4.4.1	Debugging	137

4.4.2	Software Maintenance	138
4.4.3	Reverse Engineering	138
4.4.4	Program Comprehension	139
4.4.5	Testing	139
4.4.6	Measurement	140
4.5	Tools using Program Slicing	141
4.5.1	CodeSurfer	141
4.5.2	JSlice	142
4.5.3	Unravel	142
4.5.4	HaSlicer	143
4.5.5	Other Tools	143
5	State-of-the-Art: Visualization	145
5.1	Characterizing Software Visualization	146
5.2	Taxonomies	153
5.3	Techniques	160
5.3.1	Nassi and Shneiderman Diagram	160
5.3.2	Information Murals	161
5.3.3	Graphs	163
5.3.4	Visual Metaphors	166
5.4	Applications and Tools	169
5.4.1	Program Comprehension	170
5.4.2	Software Evolution	172
5.4.3	Visualization of Program Slices	174
5.4.4	Algorithms Visualization/Animation	175
6	Verification Graphs	177
6.1	Verification Conditions	180
6.2	Labeled Control Flow Graphs	190
6.3	Verification Graphs	193
7	Assertion-based Slicing	203
7.1	Assertion-based Slicing	206
7.1.1	Postcondition-based Slicing	206
7.1.2	Precondition-based Slicing	210
7.1.3	Specification-based Slicing	211
7.2	Formalization of Assertion-based Slicing	213
7.3	Properties of Assertion-based Slicing	216
7.3.1	Removable Commands	217
7.3.2	Slicing Subprograms	219
7.3.3	Intermediate Conditions	222
7.4	Slice Graphs	223
7.5	Removing Redundant Code	230
7.6	Related Approaches	233

8	Contract-based Slicing	239
8.1	Open / Closed Contract-based Slicing	240
8.2	Contract-based Slicing: General Case	242
8.3	A Contract-based Slicing Algorithm	243
9	GamaSlicer tool	245
9.1	GamaSlicer Architecture	246
9.2	Verifying	249
9.3	Slicing and Visualizing/Animating	261
9.3.1	Assertion-based Slicing	261
9.3.2	Contract-based Slicing	269
10	Conclusion	277
A	Java Modeling Language	283
B	Satisfiability Modulo Theories	287

List of Figures

1.1	Roadmap of this document	5
1.2	Conceptual Map relating concepts and chapters	6
2.1	CFG corresponding to the program listed in 2.1	11
2.2	PDG corresponding to the program listed in 2.1	11
2.3	Components of code analysis	15
2.4	Relationship among code analysis	20
3.1	Truth tables for the connectives \neg , \wedge , \vee and \rightarrow	43
3.2	System \mathcal{N}_{FOL} for classical first-order logic	56
3.3	Abstract syntax of While^{DbC} language (x and \mathbf{p} range over sets of <i>variables</i> and <i>procedure names</i> respectively).	58
3.4	Binary Decision Tree for the function $f(x_1, x_2, x_3)$	63
3.5	Binary Decision Diagram for the function $f(x_1, x_2, x_3)$ with $x_1 = 0$, $x_2 = 0$ and $x_3 = 1$	65
3.6	Kripke Structure: Example	66
3.7	Labeled Transition System: Example	67
3.8	General architecture of a program verifier using a VCGen	68
3.9	Inference system of Hoare logic without consequence rule: system \mathcal{H}_g	71
3.10	Weakest precondition of annotated blocks	73
3.11	VC^w auxiliary function	74
3.12	Strongest postcondition of annotated blocks	76
3.13	VC^s auxiliary function	77
3.14	High level overview of Bounded Model Checking [DKW08]	82
3.15	Runtime Monitor System.	85
3.16	Test-driven Software development process.	86
3.17	ESC/Java2 architecture.	94
3.18	Frama-C architecture	96
3.19	Boogie architecture	97
3.20	VCC architecture	98
4.1	Relationships between program slicing models	118
4.2	Example system and its SDG	126

4.3	The SDG from Figure 4.2 sliced with respect to the formal-out vertex for parameter z in procedure <i>Increment</i> , together with the system to which it corresponds.	127
4.4	The DDG for the example listed in program 4.12.	134
5.1	Software Engineering Process and Software Visualization Role	147
5.2	Venn diagram showing the relationships between the various forms of software visualization [BBI98]	150
5.3	Visualization pipeline	152
5.4	Myers taxonomy	154
5.5	<i>Price et al</i> taxonomy	155
5.6	<i>Roman and Cox</i> taxonomy	156
5.7	<i>Beron et al</i> taxonomy	159
5.8	<i>Karavirta et al</i> taxonomy (parcial)	160
5.9	<i>Nassi-Shneiderman</i> diagram for Factorial program	161
5.10	Line and pixel representation of a program using <i>Ball and Eick</i> technique	162
5.11	Execution mural of message traces for an object-oriented program	163
5.12	Orthogonal Layout produced by <i>Nevron</i>	168
5.13	Force-Directed Layout produced by <i>Nevron</i>	168
5.14	Hierarchical Layout produced by <i>yFiles</i>	169
5.15	Tree Layout produced by <i>Nevron</i>	169
5.16	Radial Layout produced by <i>Nevron</i>	170
6.1	Verification condition for the UKTakesCalculation program . .	179
6.2	Verification conditions for blocks of commands with procedure call	182
6.3	LCFG for program 6.2 with respect to the specification ($y > 10, x \geq 0$)	194
6.4	Example verification graph: procedures partstep and partition	200
7.1	Definition of relation “is portion of”	213
7.2	Example slice graph (extract). Thick lines represent edges that were added to the initial CFG, corresponding to “short-cut” subprograms that do not modify the semantics of the program. These paths have the same origin and destination nodes as other longer paths corresponding to removable sequences	225
7.3	Example slice graphs	228
7.4	Propagated conditions for the program in Listing 7.8	234
7.5	Simplified conditions for the program in Listing 7.8	235
9.1	GamaSlicer architecture	247
9.2	Factorial program annotated in JML	251

9.3	Identifier Table of the Factorial program in Listing 9.1	252
9.4	AST of the Factorial program in Listing 9.1	253
9.5	Verifying the Factorial program	254
9.6	Verification Conditions for the Factorial program	255
9.7	SMT code for the verification conditions of TaxesCalculation	257
9.8	Verification Graph for the TaxesCalculation program	259
9.9	Verifying the Edge Conditions of Verification Graph for the TaxesCalculation program (1)	260
9.10	Verifying the Edge Conditions of Verification Graph for the TaxesCalculation program (2)	260
9.11	Precondition-based slicing applied to program in Listing 9.3 .	263
9.12	Specification-based slicing applied to program in Listing 9.3 .	264
9.13	Animating specification-based slicing applied to program in Listing 9.4 (Step 1)	264
9.14	Animating specification-based slicing applied to program in Listing 9.4 (Step 2)	265
9.15	Animating specification-based slicing applied to program in Listing 9.4 (Step 3)	265
9.16	Animating specification-based slicing applied to program in Listing 9.4 (Step 4)	266
9.17	Animating specification-based slicing applied to program in Listing 9.4 (Step 5)	266
9.18	Animating specification-based slicing applied to program in Listing 9.4 (Step 6)	267
9.19	Animating specification-based slicing applied to program in Listing 9.4 (Step 7)	267
9.20	Animating specification-based slicing applied to program in Listing 9.4 (Step 8 — Shortest Path)	268
9.21	Animating specification-based slicing applied to program in Listing 9.4 (Step 9 — Final Slice Graph)	268
9.22	The initial postconditions table	272
9.23	Changing postconditions table when a function call is found (1)	273
9.24	Changing postconditions table when a function call is found (2)	274
9.25	Final program with the lines to be removed highlighted . . .	275
A.1	Fragment of the Java grammar extended with JML specifica- tions	285
B.1	Fragment of the SMT-LIB grammar	289

List of Tables

3.1	Truth table for the boolean function $f(x_1, x_2, x_3)$	64
5.1	Overview of the relations between the proposed dimensions and the criteria defined in the taxonomies of <i>Roman and Cox</i> and <i>Price et al</i> respectively [MMC02].	157
9.1	Some of the patterns recognized by GamaSlicer	247
A.1	Some of JML's extension to Java expressions	284

Chapter 1

Introduction

*Every science begins as
philosophy and ends as art.*

William Durant, 1885 — 1981

The idea of focusing on a specific part of a software system for its analysis, without caring about the remaining parts composing the system, is something that pleases software engineers when they are required to understand that part of the system or even change it. It was this idea that Mark Weiser defended in his PhD thesis [Wei79] when he introduced the key concept of Program Slicing: given a slicing criterion, selecting the parts in the system that influence or are influenced by such criterion. Many other authors followed him and came up with more refined concepts of slicing: *dynamic slicing* that takes into account a particular execution of the program; *quasi-static slicing* that is an intermediate concept between the original definition of static slicing and dynamic; *hybrid slicing* that combines both static and dynamic concepts; among many others.

All these notions of slicing are based on a syntactic *slicing criterion* which consists of a statement selected from the source code (usually a line number) and a set of variables (and possibly a set of input values for such variables). But sometimes it would be interesting to go beyond the syntax and cover also the semantics aspect. Canfora et al pursued this idea and introduced the concept of *conditioned slicing* [CCLL94], characterized by the use of a condition to obtain a slice of a program. In this case, the slicing criterion consists of a statement i in the program, a set of variables V and a condition C . A conditioned slice is composed of all the statements and predicates that might affect the values of the variables in V just before the statement i is executed, when the condition C holds. This semantics-preserving approach usually results in smaller slices when compared with the traditional syntax-preserving slicing algorithms.

When the basic ideas of slicing appeared and were being developed in

the context of the areas of Program Comprehension and Maintenance, the Software Development community was still seeking for formal approaches (semantics oriented) to establishing that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not ‘go wrong’, for instance no errors occur during evaluation of expressions (the so-called *safety* behavior).

Following this thread, the eighties saw the birth of an approach called *Design by Contract* [Mey86], introduced by Bertrand Meyer, whose main idea is to include in a software system the mutual obligations and benefits between a “client” and a “supplier”. Both agree on a contract that should be expressed in the source code. This is the metaphor upon which the Design by Contract approach relies.

Applying this metaphor to a program or a procedure, the basic ideas are:

- To expect that a certain condition will be met on entry by any client module that calls it: the *precondition* P .
- To guarantee a certain property on its exit: the *postcondition* Q .
- To maintain the validity of a certain property, assumed on entry and guaranteed on exit: the *invariant* I .

Basically, a *contract* can be defined as the formalization of these obligations and benefits; one of its most relevant applications is to facilitate the *verification* of programs.

The verification problem can be reduced to the problem of checking if for each procedure in a program, the code is in accordance with its specification (or *contract*). Namely, for every procedure \mathbf{p} in a program Π with a contract (P, Q) , if P (the precondition) is true when \mathbf{p} starts, then Q (the postcondition) is invariantly true when \mathbf{p} finishes.

Along this document, the setting for the verification of programs (consisting of several procedures) will rely on the principles associated with the design by contract methodology. Each individual procedure can be verified assuming that every procedure it invokes is correct with respect to its announced specification, or contract. If this is successfully done for every procedure in a program, then the program is correct.

Technically, the soundness of this approach is based on a *mutual recursion* principle – a set of verification conditions is generated for each procedure assuming the correctness of all the procedures in the program, including itself. If all verification conditions are valid, correctness is established *simultaneously* for the entire set of procedures in the program, with all correctness assumptions dropped.

Modern program verification systems are based on algorithms that examine a program and generate a set of *verification conditions* (or *proof obligations*) that are sent to an external theorem prover for checking. If all the conditions generated from a program and its specification can be proved, then the program is guaranteed to be correct with respect to that specification.

The general purpose of this work is to explore the relation between both domains (slicing and verification), looking for mutual benefits. At first sight, it is immediate to conclude that there are two points of contact between slicing and verification: first, traditional slicing, applied a priori, may facilitate the verification of large programs. Second, and this is the main topic of this PhD work, slicing programs based on semantic, rather than syntactic, criteria, can be helpful for a number of diverse software engineering tasks, such as studying and optimizing the reuse of programs.

It was precisely this idea that Commuzi and Hart [CH96] explored when they introduced the concept of *p-slices* (predicate-slice) or *postcondition-based slices*. In this case, the slicing criterion is not anymore a statement selected from the program but a predicate (a condition that should be satisfied at the end of the execution of a procedure — a postcondition). Basically, the idea is to remove the statements in the program that do not contribute to the truth of the postcondition in the final state of the program, i.e. their presence is not required in order for the postcondition to hold (p-slices are computed using Dijkstra’s weakest precondition predicate transformer). As an extension of this work, Chung et al introduced the notions of *precondition-based slicing* and *specification-based slicing* [CLYK01], which consist in the use of a precondition, or both a precondition and a postcondition, respectively, to slice a program.

The work reported in this Ph.D. dissertation explores these ideas in the context of design by contract using specifications to slice programs.

Thesis I advocate that the use of specifications as slicing criteria helps to obtain more aggressive slices, in the sense that they are semantics-based.

1.1 Contributions and Document Structure

The first four chapters of this document are devoted to the state-of-the-art in the research areas involved in the thesis:

- In Chapter 2, a general overview of *Code Analysis* is given: techniques used nowadays to extract information from code suitable for analysis and transformation are reviewed. Also, current challenges are discussed.

- In Chapter 3, a general overview of *Program Verification* is given: the manual, fully automated, static and dynamic approaches to program verification are reviewed, giving special emphasis to the study of the Verification Condition Generators and their algorithms.
- In Chapter 4, a general overview of *Program Slicing* is given: the different notions of slicing (that have appeared since the original definition) are introduced, discussed and compared. The methods and techniques (static and dynamic) used in program slicing are also reviewed.
- In Chapter 5, a general overview of *Program Visualization* is given: different techniques used to display the information extracted from a program are reviewed. The existing taxonomies to classify the software visualization systems are also presented.

Each one of these chapters starts with basic definitions needed in the rest of the chapter, and closes with a section on applications and existing tools.

The remaining chapters are dedicated to the four main contributions of the PhD work reported here:

- Chapter 6, *Verification Graphs*
- Chapter 7, *Assertion-based Slicing*
- Chapter 8, *Contract-based Slicing*
- Chapter 9, the *GamaSlicer tool*

Chapter 6 establishes a formal basis for generating verification conditions in a way that combines forward and backward reasoning, mixing weakest precondition and strongest postcondition computations. This results in an *interactive verification conditions generator*, based on the use of *verification graphs* (control flow graphs annotated with logical assertions — called *Labeled Control Flow Graps* or LCFG for short). In addition to the user-guided generation of verification conditions, these graphs can be used for the implementation of pre-defined strategies that bring together the advantages of forward and backward reasoning; they allow for a closer relation between verification conditions and error paths, and they also support visualization applications.

An extensive study of specification-based slicing has shown that the original algorithm is not efficient and in particular it does not produce minimal slices. In Chapter 7, traditional postcondition-, precondition-, and specification-based slicing algorithms are revisited and improved (their formalization is proposed under the name of *assertion-based slicing*). A new algorithm is presented, based on the previous notion of LCFG. This new algorithm produces minimal slices (in a relative sense that will be made

clear). These definitions and algorithms, along with their application to code annotated with contracts, are discussed in this chapter.

In the same theoretical framework, the semantic slicing algorithms are extended to work at the program (interprocedural) level through the introduction of a new concept called *contract-based slicing*. Contract-based slicing enables optimizations in the context of code reuse. The aim is to bring to the interprocedural level the power of specification-based slicing. These ideas are presented in Chapter 8 and are being further developed in two master theses, one of which has already been successfully defended. Both theses contribute to bringing the ideas to the context of realistic programming languages and applications (see Chapter 10 for more details).

To help us test, evaluate, and compare the techniques introduced here with previous work, we have also developed an environment that implements all the existing semantic slicing algorithms, together with the ones introduced in this dissertation. This fourth contribution is based on generic *graph visualization and animation* algorithms that were adapted to work with *verification* and *slice* graphs (more generally, *labeled control flow graphs*). GamaSlicer is available as a web-based tool and as a desktop environment and is the topic of Chapter 9.

This document closes in Chapter 10 with a summary of the work done, summing up the novel results and their impact, as well as the identification of some trends for future work.

Figure 1.1 is a roadmap with the possible paths than can be taken through the document. Figure 1.2 contains a conceptual map that relates the different concepts mentioned and introduced in this document.

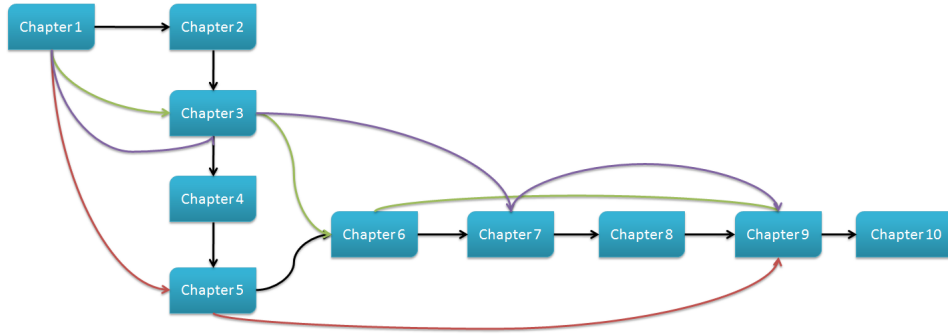


Figure 1.1: Roadmap of this document

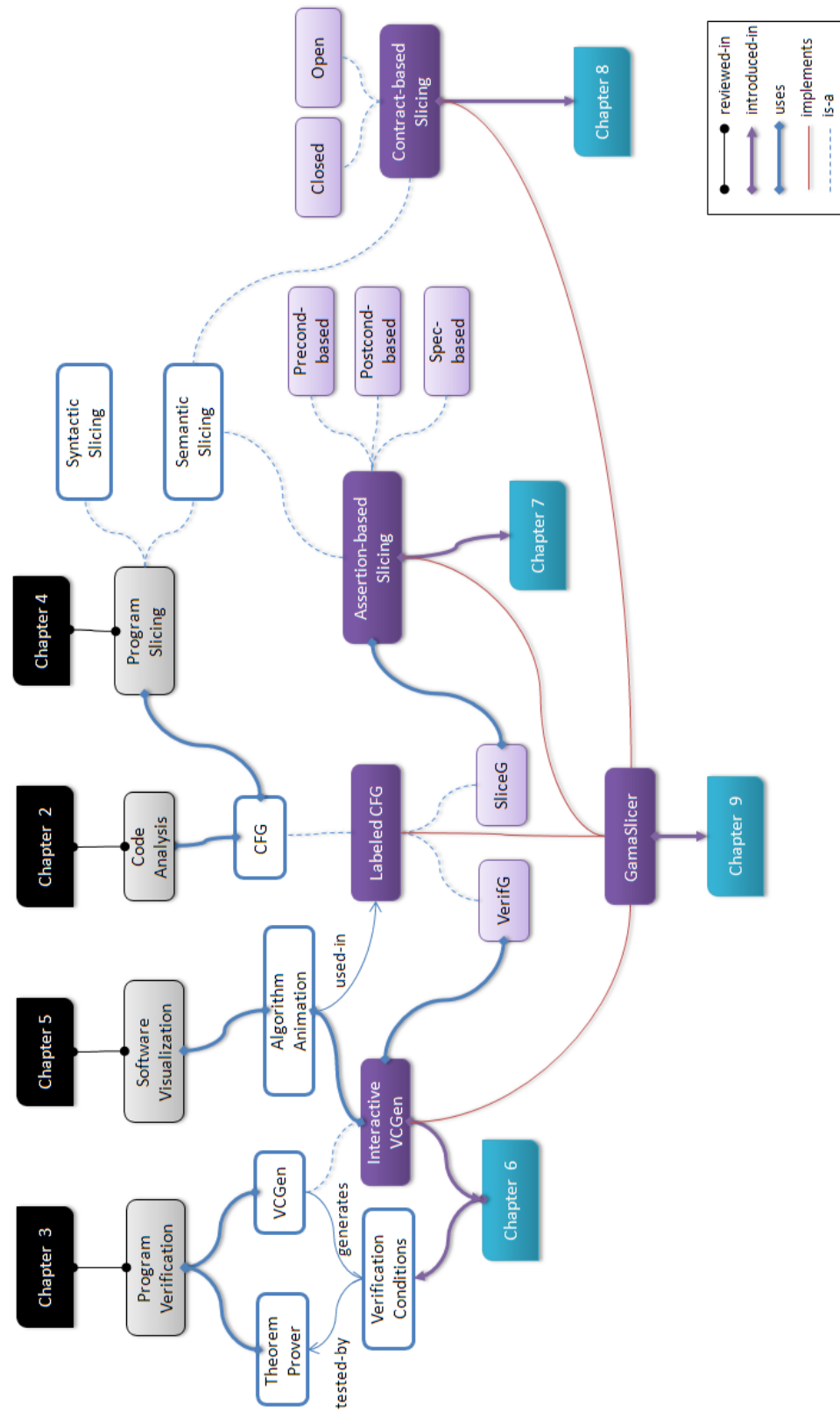


Figure 1.2: Conceptual Map relating concepts and chapters

Chapter 2

State-of-the-Art: Code Analysis

*An absolute can only be given in
an intuition, while all the rest
has to do with analysis.*

Henri Bergson, 1859 — 1941

The increasing amount of software developed in the last few years has produced a growing demand for programmers and programmer productivity to maintain it working along the years. During maintenance, the most reliable and accurate description of the behavior of a software system is its source code. However, given the complexity of modern software, the manual analysis of source code is costly and ineffective. A more viable solution is to resort to tool support. Such tools provide information to programmers that can be used to coordinate their efforts and improve their overall productivity.

In [Bin07], David Binkley presents a definition of source code analysis:

Source code analysis is the process of extracting information about a program from its source code or artifacts (e.g. from Java byte code or execution traces) generated from the source code using automatic tools. *Source code* is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form. To support dynamic analysis the description can include documents needed to execute or compile programs, such as program inputs.

The rest of this chapter will have as basis this definition of source code analysis.

At the earlier stage of compilers (when they were introduced), programmers compiled their code and then made minor adjustments (tweaks) to the

output assembly code to improve its performance. Once adjusted, future updates (that might be better made at high-level source) required one of three choices:

- re-adjusting the assembly code;
- performing the changes at the lower-level (assembly code); or
- changing the high-level source code, recompiling and forgetting the adjustments.

The final option was mostly adopted after the emergence of improved compiler technology and faster hardware.

Nowadays, modern software projects often start with the construction of models (e.g. using UML). These models can be “compiled” to a lower-level representation: source code. But this code is incomplete and thus requires that the programmers analyze the generated code and complete it. Until such models are fully executable, the source code is considered “the truth” and “the system”.

So, in both cases, code analysis is a relevant task in the life cycle of programs.

There are two kinds of code analysis: static and dynamic. In both of them, the extracted information must be coherent with the language semantics and should be disproofed from lexical concerns, focusing on abstract semantic information. This extracted information should help programmers gain insight of the source code’s meaning.

Structure of the chapter. In Section 2.1 some basic concepts of the area are presented. Section 2.2 presents the stages of a typical code analysis. In Section 2.3 some current code analysis techniques are discussed. In Section 2.4 applications of code analysis are reviewed, and Section 2.5 covers some tools for code analysis are presented.

2.1 Basic Concepts

This section introduces some basic concepts related not only with code analysis but also with the other areas covered by this thesis. These concepts are relevant to a better understanding of the remainder of this and the following chapters.

Definition 1. *An Abstract Syntax Tree (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators.*

Definition 2. A Control Flow Graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. A CFG contains a node for each statement and control predicate in the program; an edge from node i to node j indicates the possible flow of control from the former to the latter. CFGs contains special nodes labeled **Start** and **Stop** indicating the beginning and the end of the program, respectively.

There are several types of **data dependencies**: *flow dependency*; *output dependency*; and *anti-dependency*. In the context of slicing, only flow dependency is relevant.

Definition 3. A node j is flow dependent on node i if there exists a variable x such that:

- $x \in DEF(i)$;
- $x \in REF(j)$; and
- there exists a path from i to j without intervening definitions of x .

where $DEF(i)$ denotes the set of variables defined at node i , and $REF(i)$ denotes the set of variables referenced at node i .

In other words, we can say that the definition of a variable x at a node i is a *reaching definition* for node j .

Control dependency is usually defined in terms of *post-dominance*.

Definition 4. A node i in the CFG is post-dominated by a node j if all paths from i to **Stop** pass through j .

Definition 5. A node j is control dependent on a node i if and only if:

- There exists a path from i to j such for that any $u \neq i$, in that path u is post-dominated by j ; and
- i is not post-dominated by j .

Notice that if j is control dependent on i , then i has two outgoing edges (i.e., it corresponds to a predicate). Following one of the edges always results in j being executed, while taking the other edge may result in j not being executed. If the edge which always causes the execution of j is labeled with *true* (*false*, respectively), then j is control dependent on the *true* (*false*) branch of i .

Definition 6. A program path from the entry node **Start** to the exit **Stop** is a feasible path if there exist some input values which cause the path to be traversed during program execution (assuming program termination).

Let V be the set of variables of program P . A feasible path that has actually been executed for some input can be mapped onto the values the variables in V assume before the execution of each statement. Such a mapping is the referred **state trajectory**. An input to the program univocally determines a state trajectory.

Definition 7. A state trajectory of length k of a program P for input I is a finite list of ordered pairs $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \dots, (p_k, \sigma_k) \rangle$, where p_i is a statement in P , $1 \leq i \leq k$, and σ_i is a function mapping the variables in V to the values they assume immediately before the execution of p_i .

Program slices can be computed using the *Program Dependency Graph* [FOW87, HRB88] both at the intraprocedural [OO84] and the interprocedural level [KFS93b], and also in the presence of goto statements [CF94]. A program dependency graph is a program representation containing the same nodes as the CFG and two types of edges: *control dependency edges* and *data dependency edges*.

Definition 8. A program dependency graph (PDG) is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies.

The concepts hereby defined of CFG and PDG are illustrated in Figures 2.1 and 2.2 with respect to the program in Listing 2.1, which asks for a number n and computes the sum and the product of the first n positive numbers.

```

1 main() {
    int n, i, sum, product;
3   scanf("%d",&n);
    i = 1;
5   sum = 0;
    product = 1;
7   while (i <= n) {
        sum += i;
9       product *= i;
        i++;
11  }
    printf("Sum: %d\n", sum);
13  printf("Product: %d\n", product);
}
```

Listing 2.1: Program example 1: iterate sum and product

In Figure 2.1 node 7 is flow dependent on node 4 because:

- a) Node 4 defines variable **product**;
- b) Node 7 references variable **product**; and

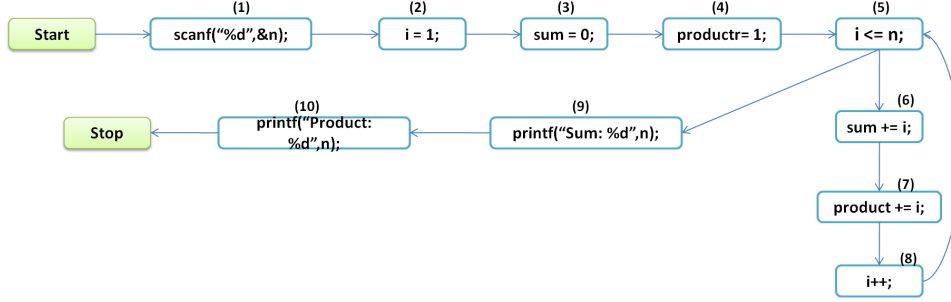


Figure 2.1: CFG corresponding to the program listed in 2.1

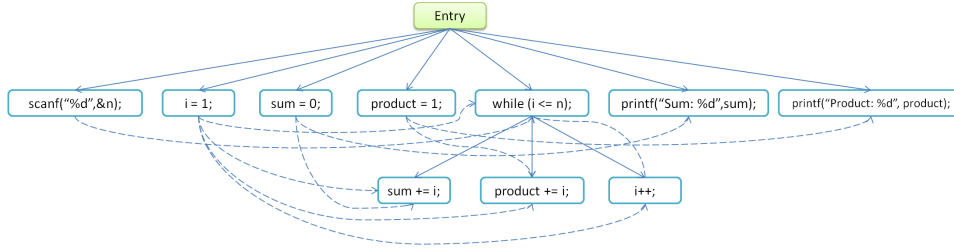


Figure 2.2: PDG corresponding to the program listed in 2.1

- c) There exists a path $4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ not containing intervening definitions of `product`.

Notice that, for the same reason, node 7 is also flow dependent on nodes 2 and 8. Also in the CFG of Figure 2.1, node 7 is control dependent on node 5 because there exists a path $5 \rightarrow 6 \rightarrow 7$ such that:

- a) Node 6 is post-dominated by node 7; and
- b) Node 5 is not post-dominated by node 7.

Figure 2.2 shows a PDG constructed according to the variant of [HRB88], where solid edges represent control dependencies and dashed edges represent flow dependencies.

Definition 9. Given a program with multiple procedures, its System Dependency Graph (SDG) is a collection of procedure-dependency graphs (one for each procedure), together with a program dependency graph for the main program. We assume that parameter passing by value-result is modeled as follows:

- a) the calling procedure copies its actual parameters to temporary variables before the call;

- b) *the formal parameters of the called procedure are initialized using the corresponding temporary variables;*
- c) *before returning, the called procedure copies the final values of the formal parameters to the temporary variables; and*
- d) *after returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.*

Then each procedure dependency graph includes vertices and edges representing call statements, parameter passing, and transitive flow dependencies due to calls.

- *A call statement is represented using a call vertex;*
- *Parameter passing is represented using four kinds of parameter vertices:*
 1. *on the calling side, parameter passing is represented by actual-in and actual-out vertices, which are control dependent on the call vertex and model copying of actual parameters to/from temporary variables;*
 2. *in the called procedure, parameter passing is represented by formal-in and formal-out vertices, which are control dependent on the procedure's entry vertex and model copying of formal parameters to/from temporary variables.*
- *Transitive dependency edges, called summary edges, are added from actual-in vertices to actual-out vertices to represent transitive flow dependencies due to called procedures.*

Actual-in and formal-in vertices are included for every global variable that may be used or modified as a result of the call and for every parameter; actual-out and formal-out are included only for global variables and parameters that may be modified as a result of the call.

Procedure dependency graphs are connected to form a SDG using three new kinds of edges:

- *a call edge is added from each call-site vertex to the corresponding procedure-entry vertex;*
- *a parameter-in edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; and*
- *a parameter-out edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.*

According to the Reps et al algorithm [RHSR94], a summary edge is added if a path of control, flow and summary edges exists in the called procedure from the corresponding formal-in vertex to the corresponding formal-out vertex. The addition of a summary edge in procedure Q may complete a path from a formal-in vertex to a formal-out vertex in Q 's PDG, which in turn may enable the addition of further summary edges in procedures that call Q .

Definition 10. A Call Graph is a directed graph that represents calling relationships between subroutines in a program. Each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g . Thus, a cycle in the graph indicates mutually recursive procedure calls.

Definition 11. A Value Dependency Graph (VDG) is a directed graph whose vertices are nodes representing computations and operand values (ports) representing values. Edges connect nodes to their operand values and ports to the computation (nodes) producing them as results. Each port is either produced by exactly one node or is a free value¹ not produced by any node.

Definition 12. Given a software system, its Module Dependency Graph (MDG) is a graph $MDG = (M, R)$ where M is the set of named modules of the system, and $R \subseteq M \times M$ is the set of ordered pairs $\langle u, v \rangle$ that represent the source-level dependencies (e.g., procedural invocation, variable access) between modules u and v of the system.

Definition 13. A XTA² Graph is a graph

$$G = \{V, E, TypeFilters, ReachableTypes\}$$

where

- $V \subseteq M \cup F\{\alpha\}$, where M is a set of methods, F is a set of fields, and α is an abstract name representing array elements;
- $E \subseteq V \times V$, is the set of directed edges;
- $TypeFilters \subseteq E \rightarrow S$, is a map from edges to a set of types S ; and
- $ReachableTypes \subseteq V \rightarrow T$, is a map from nodes to a set of resolved types T .

¹The free values of a VDG can be viewed as analogous to the free variables in a lambda term.

²XTA [QH04a] is a mechanism for implementing a dynamic reachability-based inter-procedural analysis.

The XTA graph combines call graphs and field/array accesses. A call from a method A to a method B is modeled by an edge from node A to node B . The filter set includes parameter types of method B . If B 's return type is a reference type, it is added in the filter set of the edge from B to A . Field reads and writes are modeled by edges between methods and fields, with the fields' declaring classes in the filter. Each node has a set of reachable (resolved) types.

Definition 14. A Trace Flow Graph (*TFG*) is derived from a collection of annotated CFGs. The *TFG* is a reduced “inlined” representation of the CFGs. In the *TFG*, all method invocations are replaced by expansions of the methods that they call, and the resulting graph is then reduced by the removal of all nodes that neither bear event annotations nor affect control flow.

We end this section with a concept that has proved to be extremely useful for Program Verification in recent years.

Definition 15. The Static Single Assignment (*SSA*) is a form of a program representation that exposes very explicitly the flow of data within the program. Every time a variable X is assigned a new value, the compiler creates a new version of X and the next time that variable X is used, the compiler looks up the latest version of X and uses that.

The central idea of the *SSA* is versioning. This representation is completely internal to the compiler, it is not something that shows up in the generated code nor could be observed by the debugger.

For example, for the program below (see Listing 2.2) the internal representation using the *SSA* form is shown in Listing 2.3.

```

int getValue() {
2   int a = 3;
   int b = 9;
4   int a = a + b;
   return a;
6 }
```

Listing 2.2: Program example

```

int getValue() {
2   int a_1 = 3;
   int b_1 = 9;
4   int a_2 = a_1 + b_1;
   return a_2;
6 }
```

Listing 2.3: Static Single Assignment Form of Listing 2.2

Notice that every assignment generates a new version number for the variable being modified. And every time a variable is used inside an expression, it always uses the latest version. So, the use of variable a in line 4 is modified to use a_1 .

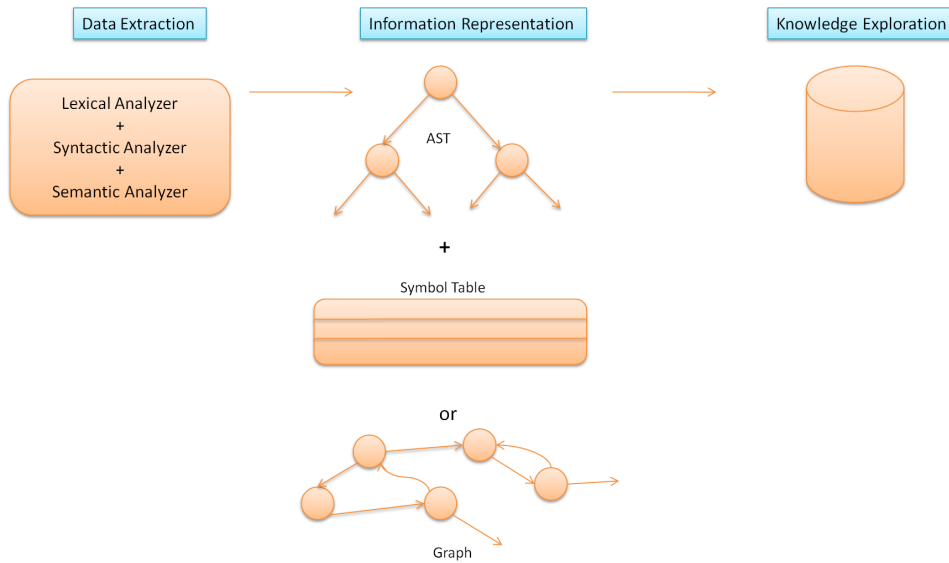


Figure 2.3: Components of code analysis

2.2 Anatomy of Code Analysis

Under the umbrella of code analysis, there are many techniques used to handle relevant static and dynamic information from a program: slicing, parsing, software visualization, software metrics, and so on. In this section, the three components needed for code analysis are described.

These three components, illustrated in Figure 2.3, are:

- Data extraction;
- Information representation; and
- Knowledge exploration.

2.2.1 Data Extraction

The process of retrieving data out of data sources for further data processing or data storage is named data extraction. Importing that data into an intermediate representation is a common strategy to make easier its analysis/transformation and possibly the addition of metadata prior to exporting to another stage in the data workflow.

In the context of code analysis this process is usually done by a syntactic analyzer, or parser, which parses the code into one or more internal representations. A parser is the part of a compiler that goes through a program and cuts it into identifiable chunks before translation, each chunk

more understandable than the whole. Basically, the parser searches for patterns of operators and operands to group the source string into smaller but meaningful parts (which are commonly called *chunks*).

Parsing is the necessary evil of most code analyzers. While not theoretically difficult, the complexities of modern programming languages, in particular those that are not LR(1) [AU72, FRJL88] and those incorporating some kind of preprocessing, significantly make harder code analysis, as will be seen in section 2.3.1.

2.2.2 Information Representation

After extracting from the code the relevant information, there is a need to represent it in a more abstract form. This is the second component of code analysis: storing the collected data into an internal representation, such that data is kept grouped in meaningful parts and the relations among them are also stored to give sense to the whole. The main goal of this phase is to abstract a particular aspect of the program into a form more suitable for automated analysis. Essentially, an abstraction is a sound, property-preserving, transformation to a smaller domain. Some internal representations are produced directly by the parser, e.g. Abstract Syntax Tree (AST), Control Flow Graph (CFG), etc, while others require the result of prior analyzes (e.g., dependency graphs require prior pointer analysis).

Many internal representations raise from the compilers area. Generally, the most common internal representation is the graph — the most widely used are the Control Flow Graph (CFG), the Call Graph, and the Abstract Syntax Tree (AST). Let us now briefly consider how each of the graphs introduced in Section 2.1 plays a role in code analysis.

The Value Dependency Graph (VDG) is another graph variant that improves (at least for some analysis) the results obtained using SSA form; VDG and SSA were both defined in section 2.1. VDG represents control flow as data flow and thus simplify analysis [WCES94].

Another relevant graph is the Dependency Graph (see Section 2.1), introduced in the context of a work with parallelizing and highly optimizing compilers [FOW87], where vertices represent the statements and predicates of the program. These graphs have since been used in other analysis [HRB88, HR92, Bal02]. A related graph, the Module Dependency Graph (MDG), used by the Bunch tool, represents programs at a coarser level of granularity. Its vertices represents modules of the system, and edges represent the dependencies between them [MMCG99].

Other sorts of graphs, also referred in the literature and defined in Section 2.1, include Dynamic Call Graphs [QH04b, PV06] intended to record an execution of a program, and XTA graphs [QH04b] which support dynamic reachability-based interprocedural analysis. These techniques are required to analyze languages such as Java that include dynamic class loading. Finally,

the Trace Flow Graph is used to represent concurrent programs [CCO01].

Finite-state Automata (FSA) are also used to represent analyzes of event-driven systems and transitions in distributed programs where they provide a formalism for the abstraction of program models [Sch02].

In real applications, it is common to combine different kinds of graphs or AST with Identifier Tables (or similar mappings) in such a way that enriches and structures the information extracted. All of the variants of graphs or other internal representations presented are actually used according to the type of analysis and the desired results of that analysis.

2.2.3 Knowledge Exploration

After organizing the data extracted into an intermediate representation that makes or transforms it into information, the third component of code analysis is aimed at knowledge inference. This process requires the inter-connection of the pieces of the information stored and their inter-relation with previous knowledge. This can be achieved using quantitative or qualitative methods. Concerning quantitative methods, resorting to program metrics is the most commonly used approach. Concerning qualitative methods, name analysis, text and data mining, and information retrieval are the most widely used. Visualization techniques are crucial for the effectiveness of that process.

According to Binkley [Bin07], the main strategies used to extract knowledge from the Intermediate Representation can be classified as follows: *static* versus *dynamic*, *sound* versus *unsound*, *flow-sensitive* versus *flow-insensitive*, and *context-sensitive* versus *context-insensitive*.

Static vs Dynamic

Static analyzes analyze the program to obtain information that is valid for all possible executions. Dynamic analyzes instrument the program to collect information as it runs. The results of a dynamic analysis are typically valid for the run in question, but offer no guarantees for other runs. For example, a dynamic analysis for the problem of determining the values of global variables could simply record the values as they are assigned. A static analysis might analyze the program to find all statements that potentially affect the global variables, then analyze the statements to extract information about the assigned values.

Dynamic analysis has the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions.

Another significant advantage of dynamic tools is the precision of the information that they provide, at least for the execution under consideration. Virtually all static analysis extract properties that are only approximations

of the properties that actually hold when the program runs. This imprecision means that a static analysis may provide information that is not accurate enough to be useful. If the static analysis is designed to detect errors (as opposed to simply extracting interesting properties), the approximations may cause the tool to report many false positives. Because dynamic analyzes usually record complete information about the current execution, it does not suffer from these problems. The trade-off, of course, is that the properties extracted from one execution may not hold in all executions.

Some techniques sit in between. They take into account a collection of initial states that, for example, satisfy a predicate.

Sound vs Unsound

A deductive system is sound with respect to a semantics if it only proves valid arguments. So, a sound analysis offers correctness guarantees.

Sound static analyzes produce information that is guaranteed to hold on all program executions; sound dynamic analyzes produce information that is guaranteed to hold for the analyzed execution alone. Unsound analyzes make no such guarantees. A sound analysis for determining the potential values of global variables might, for example, use pointer analysis to ensure that it correctly models the effect of indirect assignments that take place via pointers to global variables. An unsound analysis might simply scan the program to locate and analyze only assignments that use the global variable directly, by name. Because such an analysis ignores the effect of indirect assignments, it may fail to compute all of the potential values of global variables.

Unsound analyzes can exploit information that is unavailable to sound analysis [JR00]. Examples of this kind of information include information present in comments and identifiers. Ratiu and Deissenboeck [RD06] describe how to exploit non-structural information, such as identifiers, in maintaining and extracting the mapping between the source code and real world concepts.

It may seem difficult to understand why an engineer will be interested in an unsound analysis. However, in many cases, the information from an unsound analysis is correct, and even when incorrect, may provide a useful starting point for further investigation. Unsound analyzes are therefore often quite useful for those faced with the task of understanding and maintaining legacy code.

The most important advantages of unsound analyzes, however, are their ease of implementation and efficiency. Reconsider the two examples cited above for extracting the potential values of global variables. Pointer analysis is a complicated interprocedural analysis that requires a sophisticated program analysis infrastructure and a potentially time-consuming analysis of the entire program; locating direct assignments, on the other hand, re-

quires nothing more than a simple linear scan of the program. An unsound analysis may thus be able to analyze programs that are simply beyond the reach of the corresponding sound analysis, and may be implemented with a small fraction of the implementation time and effort required for the sound analysis. For all these reasons, unsound analysis will continue to be important.

A slightly different concept is that of *safe analysis*.

Safe static analysis means that the answer is precise on “one side”, in particular it asserts the absence of certain situations or occurrences. For example, a reaching-definitions computation can determine that certain assignments definitely do not reach a given use, but the remaining assignments may or not reach the use.

Sagiv et al [SRW02] present a static analysis technique based on a three-valued logic, capturing indecision as a third value. Thus again using reaching-definition as an example, a definition could be labeled “reaches”, “does not reach”, or “might reach”.

Flow sensitive vs Flow insensitive

Flow-sensitive analysis takes the execution order of the program’s statements into account. It normally uses some form of iterative dataflow analysis to produce a potentially different analysis result for each program point. Flow-insensitive analyzes do not take the execution order of the program’s statements into account, and are therefore incapable of extracting any property that depends on this order. They often use some form of type-based or constraint based analysis to produce a single analysis result that is valid for the entire program.

For example, given the sequence `p = &a; q = p; p = &b;`, a flow-sensitive points-to analysis can determine that *q* does not point to *b*.

In contrast, a *flow-insensitive* analysis treats the statements of a program as an unordered collection and must produce conservative results that are safe for any order. In the above example, a flow-insensitive points-to analysis must include that *q* might point to *a* or *b*. This reduction in precision comes with a reduction in computational complexity.

Context sensitive vs Context insensitive

Many programming languages provide constructs such as procedures that can be used in different contexts. Roughly speaking, a *context-insensitive* analysis produces a single result that is used directly in all contexts.

A *context-sensitive* analysis produces a different result for each different analysis context. The two primary approaches are to reanalyze the construct for each new analysis context, or to analyze the construct once (typically in the absence of any information about the contexts in which it will be used)

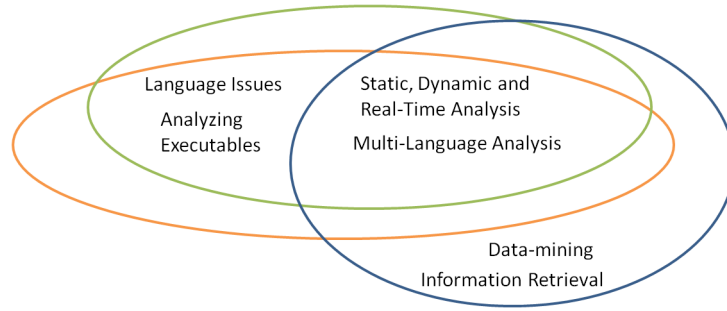


Figure 2.4: Relationship among code analysis

to obtain a single parameterized analysis result that can be specialized for each analysis context.

Context sensitivity is essential for analyzing modern programs in which abstractions (such as abstract datatypes and procedures) are pervasive.

2.3 Current Code Analysis Challenges

In this section we present the challenges that are being posed to the area of code analysis. Many of these challenges are not related with only one of the components referred in the previous section; instead, each issue affects more than one component. The relationship between these challenges and the code analysis components mentioned in the previous section are depicted in Figure 2.4 in Venn Diagram form, where: the orange color refers to the first component (data extraction); the green color refers to the second component (information representation); and the blue color refers to the third component (knowledge exploration).

2.3.1 Language Issues

In the last few years, many enhancements have been done to programming languages, with the introduction of concepts such as dynamic class loading and reflection in languages such as **Java** and **C#**.

Language reflection provides a very versatile way of dynamically linking program components. It allows to create and manipulate objects of any classes without the need to hardcode the target classes ahead of time. These features make reflection especially useful for creating libraries that work with objects in very general ways. For example, reflection is often used in frameworks that persist objects to databases, **XML**, or other external formats.

Reflection has a couple of drawbacks. One is the performance issue. Reflection is much slower than direct code when used for field and method access. A more serious drawback for many applications is that using reflection can obscure what's actually going on inside the code. Programmers expect to see the logic of a program in the source code, and techniques such as reflection that bypass the source code can create maintenance problems. Reflection code is also more complex than the corresponding direct code.

Dynamic class loading is an important feature of the **Java Virtual Machine (JVM)**. It provides the **Java** platform with the ability to install software components at runtime. It has a number of unique characteristics. First of all, lazy loading means that classes are loaded on demand and at the last moment possible. Second, dynamic class loading maintains the type safety of the **JVM** by adding link-time checks, which replace certain runtime checks and are performed only once. Finally, class loaders can be used to provide separate name spaces for various software components. For example, a browser can load applets from different web pages using separate class loaders, thus maintaining a degree of isolation between those applet classes.

Modern languages increasingly require tools for high precision source code analysis to handle only partially known behavior (such as generics in **Java**, plug-in components, reflection, user-defined types, and dynamic class loading). These features increase flexibility at runtime and impose a more powerful dynamic analysis, but compromise static analysis.

2.3.2 Multi-Language Analysis

Many software systems are heterogeneous today, i.e., they are composed of components of different programming and specification languages. Analysis using current software development tools, e.g., Integrated Development Environments (IDEs), cannot process these mixed-language systems as a whole since they are too closely related to a particular programming language, and do not process mixed-language systems across language boundaries.

For this reason, multi-language analysis grows more and more important. Even a simple **Java** program could consist of **Java**-source and -bytecode components. A larger system, e.g., a **WEB** application, joins **SQL**, **HTML**, and **Java** codes on the server site and additional languages on the client site. For example, the **Visual Studio .Net** environment merges languages such as **ASP**, **HTML**, **C#**, **J#**, and **Visual Basic**.

Listing 2.4 shows a fragment of a small **WEB** application, that illustrates such a mix of languages. This example contains an **ASP.Net** web page file. The **ASP** web page is basically an **HTML** file with some special **ASP.Net** elements and program code. When this page is requested on an **ASP** application server, the code is executed first, which results in the translated **HTML** code sent to the client. The page contains **C#** code in a script region. This code defines the event associated to the button defined in the

ASP code. The page also contains a special HTML element `<asp:Button>`, which represents a button. This element has an attribute *ID* with the value *Button1*. The ASP application server uses this *ID* to allow program code to refer to the `<asp:Button>` element and to modify it before it is sent to clients.

```

2 <%@ Page Language="C#" %>
3
4 <script runat="server">
5     protected void Button1_Click(object sender, EventArgs e) {
6         Response.Redirect("Home.aspx");
7     }
8 </script>
9
10 <html>
11     <head>
12         <script type="text/javascript">
13             function ShowModalPopup() {
14                 var modal = $find('ModalPopupExtender');
15                 modal.show();
16             }
17         </script>
18     </head>
19     <body>
20         <form id="form1" runat="server">
21             <div>
22                 <asp:Button ID="Button1" runat="server"
23                     OnClick="Button1_Click" Text="Button" />
24             </div>
25         </form>
26     </body>
27 </html>

```

Listing 2.4: Fragment of a ASP.net application

To support these mixed-language systems with automated analysis, information from all different sources ought to be retrieved and commonly processed. Only a system with a global view allows for a global correct analysis. Today’s IDEs fail in cross-language analysis. At best, they can only handle several programming languages individually.

In this context of cross-language analysis, Strein et al [SKL06] claim that the reason for this gap is the lack of a common meta-model capturing program information that is common for a set of programming languages, abstracting from details of each individual language, and is related to the source code level of abstraction, in order to allow for code analysis. The authors propose an architecture for analysis composed by three major classes: information extracting front-ends, a component meta-model (model data-structure), and analysis components.

According to the three steps described in the previous section, these three components match with the components referred for code analysis, where the common meta-model corresponds to the intermediate representation. The common meta-model captures program information in a language independent way.

Different language-specific front-ends extract information from programs written in the respective languages. They use language-specific analysis and first capture information about the program in a language specific meta-model. Information that is relevant for global analysis is also stored in the common model.

The front-ends retrieve the information represented in the common model to implement low-level analyzes (e.g. to look-up declarations). Different high-level analyzes access the common model, which represents information gained from analysis of a complete mixed-language program. Thus, concrete analyzes based on this information are language-agnostic and can handle cross-language relations.

In short, a front-end is responsible for parsing and analyzing specific languages, whereas the common model stores the relevant analysis information abstracting from language-specific details. The common meta-model is accessed by language-independent analysis. This meta-model does not need to be a union of all language concepts of all languages supported. Instead, it is sufficient to model only those language concepts that are relevant to higher-level analysis or to other languages.

Formally, each *front-end* supports a specific file type F that incorporates a set of supported languages: $F = \{L_1, L_2, \dots, L_n\}$. A file type F -specific front-end $F^{\mathcal{F}}$ is defined by a triple:

$$F^{\mathcal{F}} = (\phi^F, \{\alpha^{L_1}, \alpha^{L_2}, \dots, \alpha^{L_n}\}, \{\sigma^{L_1}, \sigma^{L_2}, \dots, \sigma^{L_n}\})$$

The front-end provides the *parsing function* ϕ^F that sorts file parts according to their languages into blocks and constructs syntax trees representing the different file blocks. For each language L of such a block, the front-end defines the *syntax mapping* α^L , that maps language-specific syntax trees AST^L to common meta-model trees AST .

For each language L the *semantic analysis function* σ^L constructs common semantic relations between nodes that are defined by the syntax mapping. σ^L is based on the common meta-model \mathcal{M} as well as a specific syntactic meta-model to handle language specificities. Through the common meta-model \mathcal{M} it can indirectly access information created by front-ends for other languages. This way, cross-language relations can be constructed for arbitrary language combinations.

The semantic relations include also dynamic relations that can actually only be computed at runtime, e.g., dynamic types in weakly or dynamically-typed languages or dynamic call targets in object-oriented languages. However, the computation of (non-trivial) dynamic properties using static analysis is generally an undecidable problem.

At least the parsing, the syntax mappings, and the semantic analysis functions need to be implemented for each new filetype. Also, it might be necessary to extend the constructs in order to capture properties of a new language.

To sum up, the key for a multi-language analysis is a common meta-model to capture the concepts of each programming language. However, as referred in subsection 2.3.1, one should parsing languages with mismatched concepts and with different principles is not an easy task, specially when

dealing with dynamic languages.

2.3.3 Static, Dynamic and Real-Time Analysis

Static analysis is usually faster than dynamic analysis but less precise. Therefore it is often desirable to retain information from static analysis for runtime verification, or to compare the results of both techniques. It would be desirable to share the same generic algorithm for static and dynamic analysis.

Martin et al describe in [MLL05] an error detection tool that checks if a program conforms to certain design rules. This system automatically generates from a query a pair of complementary checkers: a static checker that finds all potential matches in an application, and a dynamic checker that traps all matches precisely as they occur.

Slightly more sophisticated combinations often use static analysis to limit the need for instrumentation in the dynamic analysis. Path testing tools use this approach as does Martin et al’s error detection tool, where “static” results are also useful in reducing the number of instrumentation points for dynamic analysis. They report that the combination proves able to address a wide range of debugging and program queries.

Gupta et al [GSH97] present an algorithm that integrates dynamic information from a program’s execution into a static analysis. The resulting technique is more precise than static analysis and less costly than dynamic analysis.

Heuzeroth et al [HHHL03] consider the problem of reverse engineering design patterns using a more integrated combination of static and dynamic analysis. In this case, static analysis is used first to extract structures regarding potential patterns and then dynamic analysis verifies that pattern candidates have the correct behavior. Here static analysis does more than improve the efficiency of the dynamic approach. The two truly complement each other.

Closer to true integration is a combination that, in essence, iterates the two to search for test data input. This technique applies a progression of ever more complex static analyzes with search. This synergistic arrangement allow low-cost static analysis to remove “obvious” uninteresting paths. It then applies relatively naive, but inexpensive dynamic search. If more testing is needed more sophisticated static and dynamic techniques are applied. All these techniques, however, fall short of a truly integrated combination of static and dynamic techniques. Future combinations should better integrate the two.

Another kind of analysis that should be considered is real-time analysis. This is an active topic of research, and has two distinct facets: compile-time and runtime. Self-healing code³ and instrumented code are runtime

³While no consensual definition of the term “self-healing” exists, intuitively, these

examples. Here analysis is performed in real time, while the program is executing. The archetypical example of this idea is *just-in-time* compilation.

Looking forward, more such processing could be done in real-time. For instance, code coverage and memory-leak analysis might be performed, at least partially, at compile time instead of at runtime. This has the advantage of providing information about a piece of code that is the current focus of the programmer.

Other challenges in code analysis will emerge in the near future, such as the combination of source code analysis with natural language analysis; real-time verification; and improved support for user interaction (rather than being asked to make a collection of similar low-level choices, tools will ask about higher level-patterns that can be used to avoid future questioning). Code analysis tools will also need to use, edit, compile, link, and runtime information, and continue to include a combination of multiple views of a software system such as structure, behavior, and runtime snapshots.

2.3.4 Analyzing Executables

In the past years there was a considerable amount of research to develop static-analysis tools to find bugs and vulnerabilities. However, most of the effort has been on static-analysis of source code, and the issue of analyzing executables was largely ignored. In the security context, this is particularly unfortunate because source code analysis can fail to detect certain vulnerabilities due to the phenomenon known as: “What You See Is Not What You eXecute” (WYSINWYX). That is, there can be a mismatch between what a programmer intends and what is actually executed on the processor.

Thomas Reps et al [RBL06] present a number of reasons why analyzes based on source code do not provide the right level of detail for checking certain kind of properties:

1. Source level tools are only applicable when source code is available, which limits their usefulness in security applications (e.g. in analyzing code from open-source projects);
2. Analyzes based on source code typically make assumptions. This often means that an analysis does not account for certain behaviors that are allowed by the compiler;
3. Programs make extensive use of libraries, including Dynamic Linked Libraries (DLL), which may not be available in source code form. Typically, source-level analyzes are performed using code stubs that model the effects of library calls.

systems automatically repair internal faults.

4. Programs are sometimes modified subsequently to compilation, e.g. to perform optimizations or insert instrumentation code [Wal91]. They may also be modified to insert malicious code. Such modifications are not visible to tools that analyze source code.
5. The source code may have been written in more than one language. As referred in the previous subsection, this complicates the life of designers of tools that analyze source code.
6. Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places.

Thus, even if source code is available, a substantial amount of information is hidden from its analyzes, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. In order to faithfully match the behavior of the program that is actually executed, a source-level analysis tool would have to duplicate all of the choices made by the compiler and optimizer, an approach that is destined to fail.

The main goal of the work presented in [RBL06] was to recover, from executables, Intermediate Representations (IR) that are similar to those that would be available had one started from source code, but expose the platform-specific details discussed above. Specifically, the authors are interested in recovering IRs that represent the following information:

- Control Flow Graphs (CFGs) with indirect jumps resolved;
- Call Graphs with indirect calls resolved;
- Information about the program's variables;
- Sets of used, killed, and possibly.killed variables for each CFG node;
- Data dependencies (including dependencies between instructions that involve memory accesses);
- Type information (e.g. base types, pointer types, and structs).

In IR recovery, there are numerous obstacles that must be overcome. In particular, in many situations debugging information is not available. So, the authors have designed IR-recovery techniques that do not rely on debugging information being present, and are language-independent.

One of the main challenges in static analysis of low-level code is to recover information about memory access operations (e.g. the set of addresses accessed by each operation). The reasons for this difficulty are:

- While some memory operations use explicit memory addresses in the instruction (easy), others use indirect accessing via address expressions (difficult);

- Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed;
- There is no notion of type at the hardware level: address values are not intrinsically different from integer values;
- Memory accesses do not have to be aligned, so word-size address values could potentially be cobbled together from misaligned reads and writes.

As a proof of concepts exposed in [RBL06], the authors implement a set of tools: `CodeSurfer/x86`, `WPDS++` and `Path Inspector`. `CodeSurfer/x86` recovers IRs from an executable that are similar to the IRs that source code analysis tools create. `WPDS++` [KRML04] is a library for answering generalized reachability queries on *weighted pushdown systems* (WPDSs) [RSJM05]. This library provides a mechanism for defining and solving model-checking and data-flow analysis problems. Finally, `Path Inspector` is a software model checker built on top of `CodeSurfer` and `WPDS++`.

To recover the IR the authors assume that the executable that is being analyzed follows a “standard compilation model”. By this, they mean that the executable has procedures, activation records, a global data region, and a heap; it might use virtual functions and DLLs; it maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the activation records for f ; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at a fixed offset in the activation records for f ; and finally the program’s instructions occupy a fixed area of memory and are not self-modifying.

With these assumptions, this set of tools gave a major contribution concerning variable and type discovery especially for aggregates (i.e., structures and arrays). The variable and type discovery phase of `CodeSurfer/x86` recovers such information for variables that are allocated globally, locally (i.e. on the runtime stack), and dynamically (i.e. from the heap). An iterative strategy is used; with each round of the analysis, information about the program’s variables and types is refined. The memory model used is an abstraction of the concrete (runtime) address space, and has two parts:

- **Memory-regions** Although in the concrete semantics the activation records for procedures, the heap, and the memory for global data are all part of one address space, for the purpose of analysis, the address space is separated into a set of disjoint areas, which are referred to as memory-regions. Each memory-region represents a group of locations that have similar runtime properties.

- **A-loc** The second part of the memory model uses a set of proxies for variables, which are inferred for each memory-region. Such objects are called *a-locs*, which stands for “abstract locations”. In addition to the a-locs identified for each memory-region, registers represent an additional class of a-locs.

Many efforts have been made to improve the recovery of IRs through the analysis of executables. However, other aspects like dynamic languages and object-oriented programming languages still need further research.

2.3.5 Information Retrieval

In the last years, Information Retrieval (IR) has blossomed with the growth of the Internet and the huge amount of information available in electronic form.

Some applications of IR to code analysis include automatic link extraction [ZB04], concept location [MSRM04], software and website modularization [GMMS07], reverse engineering [Mar03], software reuse impact analysis [SR03, FN87], quality assessment [LFB06], and software measurement [Hoe05, HSS01].

These techniques can be used to estimate a language model for each “document” (e.g. a source file, a class, an error log, etc) and then a classifier can be used (e.g. a classifier based on the Bayesian theorem which relates the conditional and marginal probabilities of two random events) to score each. Much of this work has a strong focus on program identifiers [LMFB06]. Unlike other approaches that consider non-source documents (e.g. requirements), this approach focuses exclusively on the code. It divides each source code module into two documents: one includes the comments and the other the executable source code.

To date, the application of IR has concentrated on processing the text from source and non-source software artifacts (which can be just as important as source) using only a few developed IR techniques. Given the growing importance of non-source documents, source code analysis should in time develop new IR-based algorithms specifically designed for dealing with such documents.

2.3.6 Data Mining

The mining of software-related data repositories has recently been a very active area. Techniques such as the analysis of large amounts of data require significant computing resources and the application of techniques such as pattern recognition [PM04], neural networks [LSL96], and decision trees [GFR06], which have advanced dramatically in recent years.

Most existing techniques have been conducted by software engineering researchers, who often reuse simple data mining techniques such as associ-

ation mining and clustering. A wider selection of data mining techniques should find more general applications, removing the requirement that existing systems fit the features provided by existing mining tools. For example, API usage patterns often involve more than two API method calls or involve orders among API method calls, leaving mining for frequent item sets insufficient. Finally, the mining of API usage patterns in development environments as well as many other tasks pose requirements that cannot be satisfied by reusing existing simple miners in a black-box way.

Data mining is also being applied to software comprehension. In [KT04], the authors propose a model and associated method to extract data from C++ source code which is subsequently to be mined, and evaluates a proposed framework for clustering such data to obtain useful knowledge.

Thus, exists the demand for the adaptation or development of more advanced data mining methods.

2.4 Applications

Over the years, source-code analysis techniques have been used for many engineering tasks, facing the challenges discussed in the previous sections and many others. This section lists applications of code analysis. Only few of them will be discussed in detail.

Applications of code analysis include: architecture recovery [Sar03]; clone detection [MM01, LLWY03]; comprehension [Rug95]; debugging [FGKS91]; fault location [MX05]; middleware [ICG07]; model checking [DHR⁺07]; model-driven development [FR07]; performance analysis [WFP07]; program evolution [BR00d]; quality assessment [RBF96]; reverse engineering [CP07]; software maintenance (multiple papers in the *European Conference on Software Maintenance and Reengineering*, CSMR⁴ show this evidence); symbolic execution [KS06]; testing [Ber07, Har00]; tools and environments [Zel07]; verification [BCC⁺03]; and web application development [Jaz07, FdCHV08].

2.4.1 Debugging

Debugging and debugger tools have been a research topic that was decreasing in strength and increasing in quantity, maybe due to the ever increasing complexity of the problems imposed by more complex compiler back-ends and new language features, such as the ones previous referred: reflection and dynamic class loading, among others. Some recent debugging innovations that counter this trend include algorithmic debugging, delta debugging, and statistical debugging.

Algorithmic debugging uses programmer responses to a series of questions generated automatically by the debugger. There are two research goals

⁴<http://csmr.eu/>

for future algorithmic debuggers: first, reducing the number of questions asked in order to find the bug, and second, reducing the complexity of these questions [Sil06].

Delta debugging systematically narrows the difference between two executions: one that passes a test and one that fails [Zel01]. This is done by combining states from these two executions to automatically isolate failure causes. At present the combination is statically defined in terms of the input, but a more sophisticated combination might use dependency information to narrow down the set of potential variables and statements to be considered.

The **SOBER** tool uses statistical methods to automatically locate software faults without any prior knowledge of the program semantics [LFY⁺06]. Unlike existing statistical approaches that select predicates correlated with program failures, **SOBER** models the predicate evaluation in both correct and incorrect executions and regards a predicate as fault-relevant if its evaluation pattern in incorrect executions significantly diverges from that in correct ones. Featuring a rationale similar to that of hypothesis testing, **SOBER** quantifies the fault relevance of each predicate in a principled way.

2.4.2 Reverse Engineering

Reverse engineering is an attempt to analyze source code to determine the know-how which has been used to create it [CP07]. Pattern-matching approaches to reverse engineering aim to incorporate domain knowledge and system documentation in the software architecture extraction process. Most existing approaches focus on structural relationships (such as the generalization and association relationships) to find design patterns. However, behavioral recovery, a more challenging task, should be possible using data mining approaches such as sequential pattern discovery. This is useful as some patterns are structurally identical but differ in behavior. Dynamic analysis can be useful in distinguishing such patterns.

2.4.3 Program Comprehension

The increasing size and complexity of software systems introduces new challenges in comprehending the overall structure of programs.

In this context, program comprehension is necessary to get a deeper understanding of software applications. If they need to be changed or extended and its original documentation is missing, incomplete, or inconsistent with the implementation of the software application. Source code analysis as performed by Rigi [MTO⁺92, TWSM94] or Software Bookshelf [FHK⁺02] is one approach for program comprehension. These approaches create a source model that enables the generation of high level sequence and collaboration diagrams. Since the collaboration between different modules also depends on runtime data, dynamic tools such as **Software Reconnaissance** [EKS, WC96],

BEE++ [BGL93] or Form [SMS01] have been developed. These approaches identify the code that implements a certain feature by generating different execution traces.

For a comprehensive understanding of any software system, several complementary views need to be constructed, capturing information about different aspects of the system in question. The 4+1 Views model, introduced in [Kru95], for example, identifies four different architectural views: the *logical* view of the system data, the *process* view of the system's thread of control, the *physical* view describing the mapping of the software elements onto hardware, and the *development* view describing the organization of the software models during development. *Scenarios* of how the system is used in different types of situations are used to integrate, illustrate and validate the above views.

Chen and Rajlich [CS00] propose a semi-automatic method for feature localization, in which an analyst browses the statically derived System Dependency Graph. The SDG describes detailed dependencies among subprograms, types, and variables at the level of expressions and statements. Even though navigation on the SDG is computer-aided, the analyst still has to search for a feature's implementation.

Understanding a system's implementation without prior knowledge is a hard task for engineers in general. So along the years many code analysis models have been proposed to aid program comprehension. However, it would be desirable to have multiple models representing alternative views. For enabling slicing and abstraction mechanisms across the models, the semantic relations between them should be well-defined. It would be useful to reflect modifications in one view directly in the other views. Moreover, for program comprehension, the environment should allow the user to easily navigate between static and dynamic views, as well as between low and high level views (for instance, the user might want to select a component in one view and explore its role in the other views).

2.5 Tools

Code analysis tools can help to acquire a complete understanding of the structure, behavior and functionality of a system, or they can assist in the assessment of the impact of a modification in the system. Code analysis tools are also useful in post-maintenance testing (for example to generate cross-reference information, and to perform data flow analysis) and to produce specification and design-level documents that record for future use the knowledge gained during a maintenance operation. Under the umbrella of reverse engineering, many tools are available that support the extraction of system abstractions and design information out of existing software systems. In this section we briefly look at a representative set of such tools.

2.5.1 FxCop

FxCop [Mic08b] is an application that analyzes modules coded in assembly (targets modules included in the .NET Framework Common Language Runtime — CLR) and reports information about the modules, such as possible design, file system localization, performance, and security improvements. Many of the issues concern violations of the programming and design rules set forth in the *Design Guidelines for Class Library Developers* [Mic08a], which are the Microsoft guidelines for writing robust and easily maintainable code by using the .NET Framework.

However, the tool has a few drawbacks: it only parses the assembly code and display it in a pretty form; it analyzes the assembly code and does not infer any kind of knowledge, because its analysis is based on a set of pre-defined and fixed rules; it does not have any kind of abstraction/visualization of the intermediate representation, obtained from the parsing, to aid program comprehension; it only analyzes assembly code files one at a time, without relating them, thus restricting the analysis and comprehension of a whole system to its individual components.

2.5.2 Lint

Lint [Joh78, Dar86] is a tool to examine C programs that compile without errors, aiding to find bugs that had escaped detection.

Lint's capabilities include:

- It complains about variables and functions which are defined but not otherwise mentioned (an exception is variables which are declared through explicit `extern` statements but are never referenced).
- It attempts to detect variables that are used before being assigned. Lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a “use” since the actual use may occur at any later time, in a data dependent fashion.
- It attempts to detect unreachable portions of the programs. It will complain about unlabeled statements immediately following `goto`, `break`, `continue`, or `return` statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases `while(1)` and `for(;;)` as infinite loops. Lint also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.
- It enforces the type-checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain

binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

Obviously a drawback of the `Lint` tool is that it is limited to the C language and uses a static approach, it does not cover modern languages with the notion of object, class, reflection, or dynamic class loading.

2.5.3 CodeSonar and CodeSurfer

Both `CodeSonar` [Gra08a] and `CodeSurfer` [Gra08b] are tools from GramaTech. `CodeSonar` is a source code analysis tool that performs a whole-program, interprocedural analysis on C/C++ code and identifies complex programming bugs that can result in system crashes, memory corruption, and other serious problems. `CodeSonar` pinpoints problems at compile time that can take weeks to identify with traditional testing. The main goals of `CodeSonar` are:

- to detect and eliminate bugs early in the development cycle, when problems are easier and less expensive to fix;
- to avoid having to debug defects that can be pinpointed quickly and simply with automated analysis; and
- to catch problems that test suites miss.

`CodeSurfer`, related to `CodeSonar`, is a program-understanding tool that makes manual reviewing of code easier and faster. `CodeSurfer` does a precise analysis. Program constructs — including preprocessor directives, macros, and C++ templates — are analyzed correctly. `CodeSurfer` calculates a variety of representations that can be explored through the graphical user interface or accessed through the optional programming API. Some of its features include:

- Whole-Program Analysis: see effects between files;
- Pointer Analysis: see which pointers point to which variables and procedures;
- Call Graphs: see a complete call graph, including functions called indirectly via pointers;
- Impact Analysis: see what statements depend on a selected statement;
- Dataflow Analysis: pinpoint where a variable was assigned its value.
- Control Dependency Analysis: see the code that influences a statement's execution.

Again, the main drawback of these tools is that they are language-dependent (is this case of C/C++).

Chapter 3

State-of-the-Art: Program Verification

Trust, but verify.

Ronald Reagan, 1911 — 2004

“Computer programs can be seen as formal mathematical objects whose properties are subject to mathematical proof” [BM85]. Every program implicitly asserts a theorem to the effect that if certain input conditions are met, then the program will do what its specifications or documentation says it will.

The process of program proving has some similarities with traditional geometry in mathematics [McG82]. Geometric constructions can be compared with programs as both have the goal to achieve a particular effect. The geometric construction is followed by a (mathematical) proof that the construction will perform the intended task. Similarly, after writing a program one can prove by a (mathematical) argument that the program performs the intended task, in the sense that it respects its specification.

Program Verification is the use of formal, mathematical techniques to debug software and software specifications. The goal of Program Verification is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not ‘go wrong’, for instance, the program does not crash if exceptional conditions occur (the so-called *safety* behavior).

However, every piece of software contains errors, and computer programs rarely work properly the first time [Ltd92]. Usually, several rounds of rewriting, testing, and modifying are required before a working solution is produced [Sto96].

When a piece of source code is submitted to a compiler or the compiled

code is executed, different kinds of errors can occur in the various stages. For instance, the compiler should be capable of detecting the wrong representation of symbols, mismatch of brackets, incorrect assignment or function calls and so on. Besides this kind of syntactic errors, that result from the incorrect typing or from malformation of language constructs, the compiler should be capable of coping with semantic errors (like undeclared variables, redeclared variables, or type inconsistencies) for fully understanding the text, i.e., it should be capable of parsing the source code and interpreting it as a semantically correct program.

At execution time other kind of errors may occur (besides the syntactic and semantic ones detected at compile time) — like indexes out of range (array subscripts), division by zero, use of uninitialized variables, use of inappropriate data, and so on — that can also be detected and controlled either by extra code generated by the compiler or by the operating system. But unfortunately, many of the software errors are logical, i.e., algorithmic mistakes which result in the program failing to perform its intended task; and these are not detectable at compile or run time.

Although most software errors are relatively harmless, (e.g. a crash of a word processor), there are cases where the occurrence of errors can have serious consequences (e.g. the failure of an aircraft's flight control software). For this kind of *safety-critical software*, that can endanger human life, its correctness is a concern for everyone (developers, users, public and authorities).

The Ministry of Defense, in the United Kingdom, has defined a standard (Defense Standard 00-56) for the verification of safety critical software [MDUK96]. This standard details two basic approaches:

- The use of *formal methods* (*correct by design*).
- The *static analysis* of the code (*conformance* with the design).

This standard emphasizes the use of code analysis to guarantee that the program will perform according to its intended task. As Program Verification can give a major contribute to certifying the correctness of programs, verification techniques have lately been widely adopted to improve the confidence in a program, making it safer.

A proof of correctness should mainly detect if a program is inconsistent with respect to its assertions (that is, if it will not perform the intended task). However, a proof itself can be erroneous. Mathematicians can err in formulating proofs; the use of verification tools can help in reducing this kind of errors.

Of course, the use of software verification tools cannot guarantee the absence of errors but it can detect certain kinds of errors; sometimes it can go further by allowing the programmer to derive parts of a program automatically from logical specifications. These specifications provide a rep-

resentation of the program’s intended meaning or goal; they say what the program is supposed to do rather than how to do it. Thus they are easier to understand than the details of the program itself, and can be directly accepted or rejected depending on whether they match or fail to match the requirements.

In this chapter, our attention will be focused on Program Verification techniques, mainly on the automated ones.

Let us start by considering some of the classes of questions that the application of Program Verification techniques can help to answer. For each question, we mention a paper that discusses the question in detail, or a tool that helps to answer it.

- Will the program crash?

In [Ame02], the authors argue that the adoption of verification and validation activities in safety systems, although dominating development costs (80 percent in some cases), increase the software quality and save time/money. That is, the adoption of development methods focused on bug prevention rather than bug detection both raise quality and decrease costs. The author also highlights that a key factor of correctness by construction is the use of unambiguous programming languages that allow for a rigorous analysis.

The SPARK¹ language, an annotated subset of Ada, was proposed as an answer to this challenge of using unambiguous programming languages to build safety or security-critical software. As argued by the creator of SPARK, “the exact semantics of SPARK require software writers to think carefully and express themselves clearly; any lack of precision is ruthlessly exposed by ... its support tool, the SPARK Examiner”.

The main characteristics of the SPARK language are: logical soundness (elimination of language ambiguities); expressive power (SPARK retains the main Ada features required for writing well-engineered object-based code); security (its static semantic rules are machine-checkable using efficient analysis algorithms); and verifiability (the precise meaning of the source code allows one to reason about it in a rigorous mathematical manner).

- Does the program compute the correct result?

The KeY project [BHS07] has as main goal to contribute for the development of high quality object-oriented software, guaranteeing that the software is correct with respect to the specification.

The idea behind the KeY tool is to provide facilities for formal specification and verification of programs within a software development

¹<http://www.altran-praxis.com/spark.aspx>

platform. The formal specification is performed using the Object Constraint Language (OCL). The tool provides support for the authoring and formal analysis of OCL constraints.

- Does the program leak private information?

Jif [MZZ⁺01] is a secure-typed programming language that extends Java. It requires the programmer to label variables with information-flow security policies as part of application development. This requires the programmer to determine in advance: the entities that will handle the data in the system; the security policies (the labels) that should govern individual program variables; and the interaction of variables throughout the application (the information flow).

After the process of labeling variables with policies, the compiler tracks the correspondence between information and policies that restrict its use, enforcing security properties within the system. The Jif compiler then flags errors wherever information leaks may occur.

To help in the process of determining the security policies, the Jifclipse [HKM07] plugin for Eclipse was developed. It provides additional tools to view hidden information generated by a Jif compilation, to suggest fixes for errors, and to get more details behind an error message.

Both tools help to detect leaks of private information.

- How long does the program take to run?

Nowadays, in real-time systems, task completion at a precise time is essential. It should be checked that each real-time task is completed within a specific time frame to ensure that the program is working correctly. For this reason it is crucial to know, for each task, its Worst-Case Execution Time (WCET).

aiT [Inf09] is a tool that statically computes tight bounds for the WCET of tasks in real-time systems. It analyzes the binary executables and takes into account the intrinsic cache and pipeline behavior. This allows to be computed correct and tight upper bounds for the worst-case execution time.

The tools referred above and many others show the importance that the area of formal verification took in the last years. Formal verification techniques have also been adopted in the area of hardware industry, as errors in such systems have enormous commercial significance.

Finally, two recent examples of formal verification of operating systems include: NICTA's Secure Embedded L4 microkernel [KEH⁺09] and Green Hills Software Integrity [Gan09]. This verification provides a guarantee (a mathematical proof) that the implementation is consistent with the specification

and that the kernel is free of implementation bugs (e.g. deadlocks, livelocks, buffer overflows, arithmetic exceptions or the use of uninitialized variables).

Impact of Program Verification Over the years, Program Verification has been applied in different contexts and with different purposes. The impact of the are was due mainly to very significant improvements in the following:

- Reliability — higher assurance in program correctness by manual or automated proofs.
- Programmer productivity — feedback on errors while typing program.
- Refactoring — checking that suggested program transformations preserve program meaning (e.g. refactoring using Type Constraints; automated support for program refactoring using invariants).
- Program Optimizations — automatically transforming programs so that they take less time and less memory power.

Verification versus Validation Before proceeding to more details about software verification techniques, let us first clarify the meaning of “Software Verification”.

The definitions of *Verification* and *Validation* are often confusing and conflicting. According to the IEEE Standard Computer Dictionary [Ger91]:

- *Validation* is the confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled.
- *Verification* is the confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled.

To sum up, Validation can be defined as the process of evaluating software at the end of its development to ensure that it is free from failures and complies with its requirements. A failure is an incorrect program behavior. Often, this validation occurs through the use of different testing approaches. Design validation encompasses software validation, but goes a step further to check for proper operation of the software in its intended use environment.

Verification can be defined as the process of determining whether or not the product of a software development process fulfills the requirements established during the previous phase. Verification approaches attempt to identify product faults or errors, which give rise to failures. Software verification looks for consistency, completeness and correctness of the software and its supporting documentation. In a software development environment,

software verification is the confirmation that the output of a particular phase of development meets all of the input requirements for that phase. In this thesis we assume that the objective evidence provided by Verification implies the use of rigorous, mathematically justified arguments or processes.

Verification techniques Verification techniques can be divided into two major groups: *dynamic* verification and *static* verification.

Dynamic verification techniques refer to those that check the software's behavior during execution. Because this process is usually done during the *Testing* phase, these dynamic techniques are also known as Test or Experimentation Techniques.

Static verification techniques are those that do a physical inspection of code before its execution. The group of static verification techniques can also be divided in two groups: the *manual or semi-automatic techniques* and the *fully automated techniques*. Techniques requiring substantial manual effort, like interaction with tools that construct a proof; techniques that use refinement steps to construct programs from specifications; and techniques that require annotations from the programmer fall in the first group. Techniques such as model checking and (abstract) static analysis fall in the second group.

Structure of the chapter. Section 3.1 introduces basic concepts necessary to the remaining sections. Section 3.2 discusses the class of manual and semi-automated (static) techniques for program verification and Section 3.3 presents the fully-automated (static) techniques. Section 3.4 discusses available dynamic techniques. The chapter concludes in Section 3.5 with the presentation of some existing tools for program verification.

3.1 Basic Concepts

This section will be dedicated to basic concepts needed for the understanding of the remaining sections. Special attention is given to propositional logic, first-order logic, and Hoare logic, as these are important for the work presented in the following chapters.

We start by briefly reviewing the three basic styles of programming language semantics.

Definition 16 (Operational Semantics). *Operational Semantics is a way to give meaning to computer programs in a mathematically rigorous way. The operational semantics of a programming language describes how a valid program is interpreted a sequence of computational steps.*

In other words, defining an operational semantics is like writing an interpreter.

In the context of functional programs, the final step in a terminating sequence returns the value of the program. In general there can be many return values for a single program, because the program can be nondeterministic. Even for a deterministic program there can be many computation sequences since the semantics may not specify exactly what sequence of operations results in the return value.

Definition 17 (Denotational Semantics). *Denotational Semantics is an approach to formalizing programming languages by constructing mathematical objects (called denotations) which describe the meaning of program expressions and commands.*

In other words, denotational semantics is a bit like translating the given program to a pure functional program.

Definition 18 (Axiomatic Semantics). *Axiomatic Semantics is an approach based on mathematical logic. Axiomatic semantics define the meaning of a command in a program by describing its effect on assertions about the program state. The assertions are logical statements - predicates with variables, where the set of variables correspond to the state of the program.*

Axiomatic semantics is closely related to Hoare Logic and the Weakest Precondition Calculus. Hoare logic will be considered in detail in subsection 3.1.3.

3.1.1 Propositional Logic

Propositional logic is the basis for any logic. As any logical system, the task of describing its elements comes in three parts: its *syntax* describes what counts as a formula; its *semantics* describes its meaning; and its *proof systems* describe what is a valid proof in the logic under consideration. Given a logical language and its semantics, it is possible to establish one or more *proof systems* for it. A proof system is said to be: *sound* if every provable formula is valid; and *complete* if every valid formula is provable.

The language of Propositional Logic language consists of:

- a set of primitive symbols, usually known as *atomic assertions*, propositional letters or variables; and
- a set of operator symbols, usually known as *logical connectives* (*and*, *or*, *not*, *implication* and *absurdum*).

In the next subsections, both the syntax and semantics of propositional logic are presented.

Syntax

Propositional formulas (or propositions) are sequences of symbols from a finite alphabet defined in Definition 19, formed according to a set of rules stated in Definition 20.

Definition 19 (Alphabet of propositional logic). *The alphabet of propositional logic consists of:*

- A countable set **Prop** of propositional symbols: P, Q, R, \dots
- The logical connectives: \wedge (conjunction), \vee (disjunction), \perp (absurdum or false), \neg (negation) and \rightarrow (implication);
- Auxiliary symbols: “(”, “)”.

Definition 20 (Propositional formulas). *The set **Form** of propositional logic formulas is the smallest set of expressions such that:*

- $\mathbf{Prop} \subseteq \mathbf{Form}$
- If $A, B \in \mathbf{Form}$, then
 1. $(A \wedge B) \in \mathbf{Form}$,
 2. $(A \vee B) \in \mathbf{Form}$,
 3. $(A \rightarrow B) \in \mathbf{Form}$,
 4. $(\neg A) \in \mathbf{Form}$,
 5. $\perp \in \mathbf{Form}$.

According to this definition, an *atomic formula* has the form \perp or P . A *structured formula* is formed by combining other formulas with logical connectives.

A *well-formed formula* is thus any atomic formula, or any formula that can be built up from atomic formulas by means of operator symbols according to the rules of the grammar.

Semantics

Having described the syntax of propositional logic, we now describe the semantics which provide its meaning. The semantics or meaning of a formula ψ with propositional symbols A, B, \dots , is given by individually assigning a truth value (**T** — “true” or **F** — “false”) to A, B, \dots , and then, according to the operators, calculating the value of ψ .

The semantics are well defined due to the fact that the rules used to build propositions are unambiguous.

Definition 21 (Valuation). *A valuation is a function $\rho : \mathbf{Prop} \rightarrow \{\mathbf{F}, \mathbf{T}\}$ that assigns truth values to propositional symbols.*

One way to specify the semantics of a logical connective is by using a *truth table*. A truth table is a complete list of possible truth values of statements. Most of the formulas will have some combinations of **T**s and **F**s in their truth table columns. Some formulas will have nothing but **T**s — they are called *tautologies*. Others will have nothing but **F**s — they are called *contradictions*.

Truth tables for the connectives \neg , \wedge , \vee and \rightarrow are given in Figure 3.1.

A	$\neg A$	A	B	$A \vee B$	A	B	$A \wedge B$	A	B	$A \rightarrow B$
F	T	F	F	F	F	F	F	F	F	T
F	T	F	T	T	F	T	F	F	T	T
T	F	T	F	T	T	F	F	T	F	F
T	F	T	T	T	T	T	T	T	T	T

Figure 3.1: Truth tables for the connectives \neg , \wedge , \vee and \rightarrow

This way, the meaning of a structured formula is obtained by combining the meanings of the sub-formulas according to a specific internal operation over the truth values, that captures the intended semantics of each connective.

For instance, the meaning of the formula $P \vee Q \rightarrow P \wedge Q$ can be obtained by computing recursively the meaning of each of its sub-formulas by consulting the truth table of the corresponding connective. Having considered parentheses as part of the syntax, the following conventions to lighten the presentation of formulas are considered:

- Outermost parentheses are usually dropped.
- Parentheses dictate the order of operations in any formula. In the absence of parentheses, the following convention about precedence is adopted. Ranging from the highest to the lowest precedence, we have respectively: \neg , \wedge , \vee and \rightarrow .
- All binary connectives are right-associative.

Writing $P \vee Q \rightarrow P \wedge Q$ is equivalent to writing $(P \vee Q) \rightarrow (P \wedge Q)$ due to the precedence given to operators, and we have the truth table:

P	Q	$P \vee Q$	$P \wedge Q$	$P \vee Q \rightarrow P \wedge Q$
F	F	F	F	T
F	T	T	F	F
T	F	T	F	F
T	T	T	T	T

The notions of *propositional model* and *validity relation* offer an alternative formulation of the semantics of propositional logic [AFPMdS11].

Definition 22 (Propositional model). A *propositional model* is a set of propositional symbols $\mathcal{M} \subseteq \mathbf{Prop}$. The validity relation $\models \subseteq \mathcal{P}(\mathbf{Prop}) \times \mathbf{Form}$ is defined inductively by (\mathcal{P} denotes the powerset over P):

$$\begin{array}{lll} \mathcal{M} \models P & \text{iff} & P \in \mathcal{M} \\ \mathcal{M} \models \neg A & \text{iff} & \mathcal{M} \not\models A \\ \mathcal{M} \models A \wedge B & \text{iff} & \mathcal{M} \models A \text{ and } \mathcal{M} \models B \\ \mathcal{M} \models A \vee B & \text{iff} & \mathcal{M} \models A \text{ or } \mathcal{M} \models B \\ \mathcal{M} \models A \rightarrow B & \text{iff} & \mathcal{M} \not\models A \text{ or } \mathcal{M} \models B \end{array}$$

Definition 23 (Validity and Satisfiability of a formula). • A formula A is said to be *valid* in a model \mathcal{M} (or \mathcal{M} is said to *satisfy* A), iff $\mathcal{M} \models A$. When $\mathcal{M} \not\models A$ the formula A is said to be *refuted* by the model \mathcal{M} .

- A formula A is *satisfiable* if there exists some model \mathcal{M} such that $\mathcal{M} \models A$. It is *refutable* if some model refutes A .
- A formula A is *valid* (also called a *tautology*) if every model satisfies A . A formula A is a *contradiction* if every model refutes A .
- Given two formulas A and B , A is said to be *logically equivalent* to B ($A \equiv B$), if A and B are valid exactly in the same models.

Considering the formula used as example previously, $P \vee Q \rightarrow P \wedge Q$, and models $\mathcal{M}_1 = \{P, Q\}$ and $\mathcal{M}_2 = \{Q\}$, it is possible to observe that:

- Since $\mathcal{M}_1 \models P$ and $\mathcal{M}_1 \models Q$, we have $\mathcal{M}_1 \models P \wedge Q$ and $\mathcal{M}_1 \models P \vee Q$. This allows us to conclude that $\mathcal{M}_1 \models P \vee Q \rightarrow P \wedge Q$.
- Since $\mathcal{M}_2 \models Q$ and $\mathcal{M}_2 \not\models P$, we have $\mathcal{M}_2 \models P \vee Q$ and $\mathcal{M}_2 \not\models P \wedge Q$. Thus, we can conclude that $\mathcal{M}_2 \not\models P \vee Q \rightarrow P \wedge Q$.

Proof System

A *proof system* is defined by a set of *inference rules* used to construct derivations. A *derivation* is a sequence of sentences that explains why a given formula — the *conclusion* or *theorem* — is deducible from a set of formulas (*assumptions*). The rules that guide the construction of such derivations are called inference rules and consist of zero or more *premises* and a single conclusion. Usually, derivations are built in the form of a tree. From now on, the following notation of separating the premises from the conclusion by a horizontal line, will be used:

$$\frac{\text{prem}_1 \quad \cdots \quad \text{prem}_n}{\text{concl}}$$

This should be read as: “whenever the conditions prescribed by the premises $prem_1, \dots, prem_n$ are met, it is allowed to conclude *concl*.”

Whilst there are various proof systems for propositional logic, the most well known is *Natural Deduction*. It was first introduced by *Gerhard Gentzen* in [Gen35] (1935), and the reason for this term is described in his dissertation:

Ich wollte zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahe kommt. So ergab sich ein “Kalkül des natürlichen Schließens”.

First I wished to construct a formalism that comes as close as possible to actual reasoning. Thus arose a “calculus of natural deduction”.

There are different variants of natural deduction proof systems, but all of them have the same basic elements: basic rules; rules to *eliminate* operators; and rules to *introduce* operators. The rules hereby presented will be written in a sequent style.

Definition 24 (Sequent). *A sequent is a judgment of the form $\Gamma \vdash A$, where Γ is a set of formulas (called the context) and A is a formula (called the conclusion of the sequent).*

A sequent $\Gamma \vdash A$ is read as “ A can be deduced from the set of assumptions Γ ” or “ A is a consequence of Γ ”.

Rules of the proof system The rules are grouped in three categories as follows.

Please notice that the formulas A , B , and C occurring in the rules are in fact *meta-variables* that can be replaced by concrete formulas to obtain specific instances of each rule (see Definition 26).

- **Basic Rules**

$$(Ax) \frac{}{\Gamma, A \vdash A}$$

This rule, called *axiom rule* is the only rule with no premises and is the base case for the inductively defined set of derivations. The axiom rule simply asserts that deduction subsumes inclusion.

$$(RAA) \frac{\Gamma, \neg A \vdash \perp}{\Gamma \vdash A}$$

The *reductio ad absurdum* (RAA or *reduction to the absurd*) is a form of argument that enables reasoning by contradiction: a formula A can

be proved by establishing that assuming $\neg A$ leads to a contradiction (that is, a proposition is proven true by proving that it is impossible for it to be false).

- **Introduction rules**

Inference rules that introduce a logical connective in the conclusion are known as *introduction rules*. Introduction rules for a logical operator \otimes express the conditions for the truth of a sentence P containing \otimes as main operator and whose premises assert the truth of immediate sub-formulas of P .

$$(I_{\rightarrow}) \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad (I_{\neg}) \frac{\Gamma, A \vdash \perp}{\Gamma \vdash \neg A}$$

These two rules modify the assumption sets used in the conclusion and in the premises.

The (I_{\rightarrow}) rule states that proving an implication corresponds to proving its conclusion assuming the antecedent as an additional assumption.

The (I_{\neg}) rule states that proving the negation of A corresponds to reaching a contradiction assuming A .

$$(I_{\wedge}) \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The (I_{\wedge}) rule states that if one can derive both A and B then $A \wedge B$ is true.

$$(I_{\vee 1}) \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad (I_{\vee 2}) \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

These rules state that if one of A **or** B is derivable, then so is $A \vee B$.

- **Elimination rules**

Elimination rules are dual to introduction rules, and describe how to de-construct information about a compound proposition into information about its constituents.

$$(E_{\rightarrow}) \frac{\Gamma \vdash A \quad \Gamma \vdash A \rightarrow B}{\Gamma \vdash B}$$

This rule, also known as *modus ponens* (latin words for *the way that affirms by affirming*) captures the conditional nature of implication: if one is able to deduce both an implication $A \rightarrow B$ and its antecedent A from a given set of assumptions Γ , then the consequent B can also be deduced from the same set of assumptions.

An example of an argument that fits the form *modus ponens*:

If today is Monday (P), then (\rightarrow) I will go to work (Q).

Today is Monday (P).

Therefore, I will go to work (Q).

$$(E_{\neg}) \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash B}$$

This rule states the principle of explosion, also known as *ex falso sequitur quodlibet*, according to which “anything follows from a contradiction”. That is, once a contradiction has been asserted, any proposition (or its negation) can be inferred from it.

The rule can be read as: “If one claims something is both true (A) and not true ($\neg A$), one can logically derive *any* conclusion (B)”.

$$(E_{\wedge i}) \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_i} \quad i \in \{1, 2\}$$

These rules, known also as *simplification*, state that if the conjunction $A_1 \wedge A_2$ is true, then A_1 is true and A_2 is true.

An example in natural language:

It is raining (A_1) and (\wedge) it is pouring (A_2).

Therefore it is raining (A_1) and it is also pouring (A_2).

$$(E_{\vee}) \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

This rule, also known as *disjunction elimination*, states that if A or B is true, and A is a consequence of C , and B is a consequence C , then we may infer C . That is, since at least one of the statements A and B is true, and since either of them is sufficient to entail C , then C is certainly true.

An example in natural language:

It is true that either I am inside (A) or I am outside (B). It is also true that if I am inside, I have my wallet on me ($C, A \vdash C$). It is also true that if I am outside, I have my wallet on me ($B \vdash C$).

Therefore it follows that I have my wallet on me (C).

Taking into account these sets of rules, we are now able to define the proof system \mathcal{N}_{PL} of natural deduction.

Definition 25 (Proof system \mathcal{N}_{PL}). *The proof system \mathcal{N}_{PL} of natural deduction for propositional logic is defined by the sets of Basic rules, Introduction rules and Elimination rules presented previously.*

Definition 26 (Instance of an inference rule). *An instance of an inference rule is obtained by replacing all occurrences of each meta-variable by a phrase in its range.*

Definition 27. *A derivation or proof in \mathcal{N}_{PL} is a finite tree whose leaves are propositional formulas, and which is built by using the rules of \mathcal{N}_{PL} .*

Example 1. *The following proof shows that $(P \wedge Q) \rightarrow R \vdash P \rightarrow (Q \rightarrow R)$ is derivable in \mathcal{N}_{PL} .*

$$\begin{array}{c}
 \begin{array}{c}
 (Ax) \frac{}{(P \wedge Q) \rightarrow R, P, Q \vdash (P \wedge Q) \rightarrow R} \\
 (E \rightarrow) \frac{}{(P \wedge Q) \rightarrow R, P, Q \vdash (P \wedge Q) \rightarrow R}
 \end{array}
 \quad
 \begin{array}{c}
 (Ax) \frac{}{(P \wedge Q) \rightarrow R, P, Q \vdash P} \quad (Ax) \frac{}{(P \wedge Q) \rightarrow R, P, Q \vdash Q} \\
 (I \wedge) \frac{}{(P \wedge Q) \rightarrow R, P, Q \vdash P \wedge Q}
 \end{array} \\
 \hline
 (I \rightarrow) \frac{(P \wedge Q) \rightarrow R, P, Q \vdash R}{(P \wedge Q) \rightarrow R, P \vdash Q \rightarrow R} \\
 (I \rightarrow) \frac{}{(P \wedge Q) \rightarrow R \vdash P \rightarrow (Q \rightarrow R)}
 \end{array}$$

See for instance [AFPMdS11] for proofs of *soundness* and *completeness* with respect to this proof system.

3.1.2 First-order Logic

In propositional logic, it is not possible to express assertions about elements of a structure. This is mainly due to its relative mathematical simplicity. First-order logic is richer than propositional logic and still enjoys some interesting mathematical properties. It allows one to reason about properties that are shared by different objects, by allowing the propositional symbols to have arguments ranging over elements of structures. In addition to specifying the meaning of predicate symbols, an interpretation must specify a non empty set, known as the domain of discourse, as a range for quantifiers. Thus, a statement of the form $\exists x. \text{printer}(x)$ is said to be true, under a particular interpretation, if there is some object in the domain of discourse of that interpretation that satisfies the predicate that the interpretation uses to assign meaning to the symbol *printer*, and $\forall x. \text{printer}(x)$ is true if every such object satisfies the predicate.

Essentially, first-order logic is a powerful notation that extends propositional logic with predicates and quantification. A predicate is a function that returns either *true* or *false*. If *printer* is a predicate of arity one and *ancestor* is a predicate of arity two, then the formula *printer*(*x*) means that *x* is a printer, and the formula $\forall x. \forall y. ((\text{ancestor}(x, \text{Mary}) \wedge \text{ancestor}(y, x)) \rightarrow \text{ancestor}(y, \text{Mary}))$ means that if *y* is an ancestor of *x*, who is an ancestor of *Mary*, then *y* is also *Mary*'s ancestor.

This section briefly introduces the basic notions of first-order logic. Similarly to propositional logic, the *syntax* of the first-order logic language will first be presented, then its *semantics*, and finally a *proof system*.

Syntax

The language of first-order logic, unlike from natural languages, is completely formal, so that it can be mechanically determined whether a given expression is legal.

There are two types of valid expressions in the language: *terms*, which represents objects, and *formulas*, which express properties that can be either true or false. The terms and formulas are strings of *symbols* which together form the *alphabet* of the language. It is common to divide the symbols of the alphabet into *logical symbols*, which always have the same meaning, and *non-logical symbols*, whose meaning varies according to the interpretation.

Definition 28 (Alphabet of first-order logic — Logical symbols). *The alphabet of logical symbols of first-order logic consists of:*

- Logical connectives: \wedge (*and*), \vee (*or*), \perp (*absurdum or false*), \neg (*not*), \rightarrow (*implication*), \forall (*universal quantifier*) and \exists (*existential quantifier*);
- Auxiliary symbols: “(”, “)”.
- Variables: a countable set \mathcal{X} of variables that represent arbitrary elements of an underlying domain. Let x, y, z, \dots range over \mathcal{X} .

The non-logical symbols represent predicates (relations), functions and constants of the domain of discourse.

Definition 29 (Alphabet of first-order logic — Non-logical symbols). *The alphabet of non-logical symbols of first-order logic consists of:*

- Constants: a countable set \mathcal{C} of constants that represent specific elements of an underlying domain. Let a, b, c, \dots range over \mathcal{C} . Examples: *Mary*, *3*, etc.
- Functions: a countable set \mathcal{F} of function symbols. Let f, g, h, \dots range over \mathcal{F} . Examples: *fatherOf* of arity 1, *colorOf* of arity 1.
- Predicates: a countable set \mathcal{P} of predicate symbols.
Let P, Q, R, \dots range over \mathcal{P} . Examples: *greater* of arity 2, *green* of arity 1.

These three sets are disjoint and they constitute the vocabulary $\mathcal{V} = \mathcal{C} \cup \mathcal{F} \cup \mathcal{P}$.

The next step is to define the terms and formulas of first-order logic.

Definition 30 (Terms). *The set of terms of a first-order language over a vocabulary \mathcal{V} is inductively defined by the following rules:*

- Every constant $c \in \mathcal{C}$ and every variable $x \in \mathcal{X}$ is a term.

- If t_1, \dots, t_n are terms and f is a function of arity $n > 0$, then $f(t_1, \dots, t_n)$ is a term.

Only expressions using these two rules are terms. No expression involving predicate symbols is a term.

Example 2. Let \mathcal{L} be the following first-order language for arithmetic where: $\mathcal{C} = \{0\}$, $\mathcal{F} = \{\text{Succ}, +, -\}$ and $\mathcal{P} = \{<\}$. The symbol Succ has arity 1 and the symbols $+$, $-$ and $<$ have arity 2. Then the following are terms:

- $\text{Succ}(0)$
- $+(\text{Succ}(0), \text{Succ}(\text{Succ}(0)))$
- $<(x_1, 0)$

Definition 31 (Formulas). The set of formulas $\mathbf{Form}_{\mathcal{V}}$ of a first-order language over a vocabulary \mathcal{V} is inductively defined by the following rules:

- \perp is a formula.
- If t_1, \dots, t_n are terms and P is a predicate symbol of arity n , then $P(t_1, \dots, t_n)$ is a formula.
- If $\psi, \phi \in \mathbf{Form}_{\mathcal{V}}$, then:
 - $(\neg\phi) \in \mathbf{Form}_{\mathcal{V}}$
 - $(\phi \wedge \psi) \in \mathbf{Form}_{\mathcal{V}}$
 - $(\phi \vee \psi) \in \mathbf{Form}_{\mathcal{V}}$
 - $(\phi \rightarrow \psi) \in \mathbf{Form}_{\mathcal{V}}$
 - $(\forall x.\phi) \in \mathbf{Form}_{\mathcal{V}}$
 - $(\exists x.\phi) \in \mathbf{Form}_{\mathcal{V}}$

An *atomic formula* is either \perp or $P(t_1, \dots, t_n)$.

Similarly to propositional logic, parentheses dictate the order of operations in any formula. In the absence of parentheses we have, ranging from the highest precedence to the lowest: $\neg, \wedge, \vee, \rightarrow$. Finally we have that \rightarrow binds more tightly than \forall and \exists .

Using the first-order language described previously, $<(0, \text{Succ}(0))$ is an atomic formula. And the following is a formula:

$$\forall x. \forall y. (<(x, y) \rightarrow \exists z. <(z, y + x - z))$$

In a quantified formula $\forall x.\phi$ or $\exists x.\phi$, x is a *quantified variable* and ϕ is its scope of quantification. Occurrences of a quantified variable in a given scope are said to be *bound*, while the occurrences that are not bound are said to be *free*. For example, in the formula $\exists x.P(x, y)$, x is bound but y is free.

Definition 32 (Free variables). *The set of free variables of a formula φ , $\mathbf{FV}(\varphi)$, is defined inductively as follows:*

- If φ is an atomic formula then $\mathbf{FV}(\varphi)$ is the set of all variables occurring in φ .
- If φ is $\neg\phi$, then $\mathbf{FV}(\varphi) = \mathbf{FV}(\phi)$.
- If φ is $\phi \wedge \psi$ or $\phi \vee \psi$ or $\phi \rightarrow \psi$, then $\mathbf{FV}(\varphi) = \mathbf{FV}(\phi) \cup \mathbf{FV}(\psi)$.
- If φ is $\forall x.\phi$ or $\exists x.\phi$, then $\mathbf{FV}(\varphi) = \mathbf{FV}(\phi) \setminus x$.

Definition 33 (Bound variables). *The set of bound variables of a formula φ , $\mathbf{BV}(\varphi)$, is defined inductively as follows:*

- If φ is an atomic formula then $\mathbf{BV}(\varphi) = \emptyset$.
- If φ is $\neg\phi$, then $\mathbf{BV}(\varphi) = \mathbf{BV}(\phi)$.
- If φ is $\phi \wedge \psi$ or $\phi \vee \psi$ or $\phi \rightarrow \psi$, then $\mathbf{BV}(\varphi) = \mathbf{BV}(\phi) \cup \mathbf{BV}(\psi)$.
- If φ is $\forall x.\phi$ or $\exists x.\phi$, then $\mathbf{BV}(\varphi) = \mathbf{BV}(\phi) \cup \{x\}$.

For example, in $\forall x. \forall y. (P(x) \rightarrow Q(x, f(x), z))$, $\mathbf{BV} = \{x, y\}$ $\mathbf{FV} = \{z\}$.

Definition 34 (Sentence, Universal Closure and Existential Closure). *A formula of first-order logic with no free variables is called a sentence. If $\mathbf{FV}(\phi) = \{x_1, \dots, x_n\}$, the universal closure of ϕ is the formula $\forall x_1, \dots, x_n.\phi$ and the existential closure of ϕ is the formula $\exists x_1, \dots, x_n.\phi$.*

An important notion in a first-order language is the substitution of a term for a free variable in a term or formula.

Definition 35 (Substitution in terms). *Let s, t be terms and x a variable. The result $s[t/x]$ of replacing in s all free instances of x by t is defined recursively by the following rules:*

$$\begin{aligned} y[t/x] &= \begin{cases} y & y \neq x \\ t & \text{otherwise} \end{cases} \\ c[t/x] &= c \\ f(t_1, \dots, t_n)[t/x] &= f(t_1[t/x], \dots, t_n[t/x]) \end{aligned}$$

Definition 36 (Substitution in a formula). *Let ϕ be a formula, t a term and x a variable. The result of replacing all free instances of x by t in ϕ is defined recursively by the following rules:*

$$\begin{aligned}
\perp[t/x] &= \perp \\
P(t_1, \dots, t_n)[t/x] &= P(t_1[t/x], \dots, t_n[t/x]) \\
(\neg\phi)[t/x] &= \neg(\phi[t/x]) \\
\phi \otimes \psi[t/x] &= (\phi[t/x]) \otimes (\psi[t/x]) \\
Qy.\phi[t/x] &= \begin{cases} Qy.\phi[t/x] & \text{if } y \neq x \\ Qy.\phi & \text{otherwise} \end{cases}
\end{aligned}$$

where $\otimes \in \{\wedge, \vee, \rightarrow\}$ and $Q \in \{\forall, \exists\}$.

The above definition is not perfect. To understand why, let ϕ be the valid formula $\exists x.(x = y)$ over the signature $(\{0, 1\}, \{+, \times, =\})$ of arithmetic. If t is the term $x + 1$, the formula $\phi[t/y] = \exists x.(x = x + 1)$, which will be false in many interpretations. The problem is that the free variable x of t became bound during the substitution. We say that the occurrence of x in t has been *captured* by the quantifier.

Substitutions in which some variable in the substituted term t becomes bound will not behave as expected, in the sense that they can change the meaning (and the truth value) of the formula in which they occur. The following definition delineates the situations in which the substitution can be performed in a safe way.

Definition 37 (Free term for a variable in a formula). *The notion of a term t free for a variable x in a formula ϕ is defined inductively as follows:*

- If ϕ is atomic, then t is free for x in ϕ .
- If ϕ is $\neg\varphi$ and t is free for x in φ , then t is free for x in ϕ .
- If ϕ is $\varphi \otimes \psi$ and t is free for x in both φ and ψ , then t is free for x in ϕ ($\otimes \in \{\wedge, \vee, \rightarrow\}$).
- If ϕ is either $\forall x.\psi$ or $\exists x.\psi$ and either
 - $x \notin \mathbf{FV}(\psi)$, or
 - $y \notin \mathbf{Vars}(t)$ and t is free for x in ψ ,
 then t is free for x in ϕ .

Rather than giving a version of Definition 36 that avoids variable capture, we will restrict its application to the situations in which t is free for x in ϕ . In practice this requirement can be met by renaming bound variables before applying the substitution.

The definition of substitution has several aspects in common with rules of inference. It is entirely syntactic and has some limitations on when it can be applied. These limitations are necessary because of the interaction between free and bound variables involved.

Semantics

Given a first-order language, the semantics of formulas is obtained by interpreting the function, constant and predicate symbols of the language (the vocabulary) and assigning values to the free variables. For this, one needs to introduce the concept of *first-order structure*.

Definition 38 (Structure). *Given a vocabulary \mathcal{V} , a \mathcal{V} -structure \mathcal{M} is a pair (D, I) where D is a non-empty set called the interpretation domain, and I is an interpretation function that assigns constants, functions and predicates over D to the symbols of \mathcal{V} as follows:*

- For every constant $c \in \mathcal{V}$, $I(c)$ is an element of D .
- For every function symbol $f \in \mathcal{V}$ of arity $n > 0$, $I(f) \in D^n \rightarrow D$.
- For every predicate symbol $p \in \mathcal{V}$ of arity $n > 0$, $I(p) \in D^n \rightarrow \{\mathbf{T}, \mathbf{F}\}$. Predicate symbols of arity 0 are interpreted as truth values.

Introduced the notion of \mathcal{V} -structure, it is time to define the semantics of formulas. However, the previous definition does not take into account the formulas containing free variables. The truth of that formula will generally depend on the specific assignment of values from the domain \mathcal{M} to the variables. So, the semantics of a formula should be defined as a function from the set of assignments of values in D to the variables, to the set $\{\mathbf{T}, \mathbf{F}\}$ of truth values. First, it is necessary to introduce the notion of *assignment*.

Definition 39 (Assignment). *Given a domain D , an assignment is a function $\alpha : \mathcal{X} \rightarrow D$ from the set of variables to D . The set of all assignments for a domain D is represented by Σ_D .*

The semantics of first-order logic formulas can now be presented. Terms are interpreted as elements of the interpretation domain, and formulas as truth values, based on a \mathcal{V} -structure and an assignment.

Definition 40 (Semantics of terms). *Let $\mathcal{M} = (D, I)$ be a \mathcal{V} -structure, α an assignment, and $t \in \mathbf{Term}_{\mathcal{V}}$. The value of t with respect to \mathcal{M} and α , written $\llbracket t \rrbracket_{\mathcal{M}}(\alpha)$, is given by the function $\llbracket t \rrbracket_{\mathcal{M}} : \Sigma_D \rightarrow D$, defined below:*

$$\begin{aligned} \llbracket x \rrbracket_{\mathcal{M}}(\alpha) &= \alpha(x) \\ \llbracket c \rrbracket_{\mathcal{M}}(\alpha) &= I(c) \\ \llbracket f(t_1, \dots, t_n) \rrbracket_{\mathcal{M}}(\alpha) &= I(f)(\llbracket t_1 \rrbracket_{\mathcal{M}}(\alpha), \dots, \llbracket t_n \rrbracket_{\mathcal{M}}(\alpha)) \end{aligned}$$

Definition 41 (Semantics of formulas). *Let $\mathcal{M} = (D, I)$ be a \mathcal{V} -structure, α an assignment, and $\phi \in \mathbf{Form}_{\mathcal{V}}$. The value of ϕ with respect to \mathcal{M} and α , written $\llbracket \phi \rrbracket_{\mathcal{M}}(\alpha)$, is given by the function $\llbracket \phi \rrbracket_{\mathcal{M}} : \Sigma_D \rightarrow \{\mathbf{T}, \mathbf{F}\}$, defined below:*

$$\begin{aligned}
\llbracket \perp \rrbracket_{\mathcal{M}(\alpha)} &= \mathbf{F} \\
\llbracket P(t_1, \dots, t_n) \rrbracket_{\mathcal{M}(\alpha)} &= I(P)(\llbracket t_1 \rrbracket_{\mathcal{M}(\alpha)}, \dots, \llbracket t_n \rrbracket_{\mathcal{M}(\alpha)}) \\
\llbracket \neg \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{F} \\
\llbracket \phi \wedge \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} \text{ and } \llbracket \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} \\
\llbracket \phi \vee \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha)} \text{ or } \llbracket \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} \\
\llbracket \phi \rightarrow \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{F} \text{ or } \llbracket \psi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} \\
\llbracket \forall x. \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha[x \mapsto a])} = \mathbf{T} \text{ for all } a \in D \\
\llbracket \exists x. \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T} &\text{ iff } \llbracket \phi \rrbracket_{\mathcal{M}(\alpha[m \mapsto a])} = \mathbf{T} \text{ for some } a \in D
\end{aligned}$$

where $f[x \mapsto y]$ is a function defined as follows:

$$(f[x \mapsto y])(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

Notice that universal and existential quantifications can be seen as generalizations of the conjunction and disjunction connectives as follows:

$$\begin{aligned}
\llbracket \forall x. \phi \rrbracket_{\mathcal{M}(\alpha)} &= \bigwedge_{a \in D} \llbracket \phi \rrbracket_{\mathcal{M}(\alpha[x \mapsto a])} \\
\llbracket \exists x. \phi \rrbracket_{\mathcal{M}(\alpha)} &= \bigvee_{a \in D} \llbracket \phi \rrbracket_{\mathcal{M}(\alpha[x \mapsto a])}
\end{aligned}$$

Now, we are able to define the concepts of *validity*, *satisfaction* and *model* in first-order logic.

Definition 42 (Satisfaction and Model). *Let \mathcal{V} be a vocabulary, \mathcal{M} a \mathcal{V} -structure, and ϕ a formula. One says that:*

- \mathcal{M} satisfies ϕ with α , denoted by $\mathcal{M}, \alpha \models \phi$, iff $\llbracket \phi \rrbracket_{\mathcal{M}(\alpha)} = \mathbf{T}$.
- \mathcal{M} satisfies ϕ (or ϕ is valid in \mathcal{M}), denoted by $\mathcal{M} \models \phi$, iff for every assignment α , $\mathcal{M}, \alpha \models \phi$. \mathcal{M} is said to be a model of ϕ .

Definition 43 (Satisfiability and Validity). • *A formula ϕ is satisfiable if there exists some \mathcal{V} -structure \mathcal{M} such that $\mathcal{M} \models \phi$ and it is valid, denoted by $\models \phi$, if $\mathcal{M} \models \phi$ for every structure \mathcal{M} .*

- *A formula ϕ is unsatisfiable (or a contradiction) if it is not satisfiable, and refutable if it is not valid.*

Proof System

The proof system presented in what follows is the natural deduction system for (classical) first-order logic in sequent style. The difference between this proof system and the one presented for propositional logic lies in the rules introduced to deal with the quantifiers.

Rules of the proof system. The system is composed of the *Basic Rules*, *Introduction Rules* and *Elimination Rules* introduced in subsection 3.1.1, together with the following *Introduction and Elimination rules for Quantifiers*.

- **Introduction rules for Quantifiers**

$$(I_{\forall}) \frac{\Gamma \vdash \phi[y/x]}{\Gamma \vdash \forall x. \phi} \text{ (a)}$$

(a) *Restriction:* y must not occur free in either Γ or ϕ .

What this rule means is that if ϕ holds for an arbitrary element y of the universe, then we can conclude that $\forall x. \phi$, i.e., one can generalize it to all elements. The restriction ensures precisely that y represents an arbitrary element.

$$(I_{\exists}) \frac{\Gamma \vdash \phi[t/x]}{\Gamma \vdash \exists x. \phi}$$

This rule (I_{\exists}) states that if the property ϕ holds for some element t in the universe, then one can deduce that there exists an element in the universe that has the property ϕ .

- **Elimination rules for quantifiers**

$$(E_{\forall}) \frac{\Gamma \vdash \forall x. \phi}{\Gamma \vdash \phi[t/x]}$$

This rule (E_{\forall}) states that if $\forall x. \phi$ can be deduced, then ϕ holds for any term.

$$(E_{\exists}) \frac{\Gamma \vdash \exists x. \phi \quad \Gamma, \phi[y/x] \vdash \theta}{\Gamma \vdash \theta} \text{ (b)}$$

(b) *Restriction:* y must not occur free in Γ , ϕ or θ .

In this rule (E_{\exists}), the second premise states that θ can be deduced if ϕ is assumed to hold for an unspecified term. The first premise states that such a term exists, thus θ can in fact be deduced with no further assumptions.

Definition 44. The proof system \mathcal{N}_{FOL} of natural deduction for first-order logic is defined by the set of basic, introduction, and elimination rules summarized in Figure 3.2.

Similarly to propositional logic, this system is *sound* and *complete* (see for instance [AFPMdS11]).

$$\begin{array}{c}
(\text{Ax}) \frac{}{\Gamma, \phi \vdash \phi} \quad (\text{I}_{\rightarrow}) \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi} \quad (\text{E}_{\rightarrow}) \frac{\Gamma \vdash \phi \quad \Gamma \vdash \phi \rightarrow \psi}{\Gamma \vdash \psi} \\
\\
(\text{I}_{\neg}) \frac{\Gamma, \phi \vdash \perp}{\Gamma \vdash \neg \phi} \quad (\text{E}_{\neg}) \frac{\Gamma \vdash \phi \quad \Gamma \vdash \neg \phi}{\Gamma \vdash \psi} \quad (\text{RAA}) \frac{\Gamma, \neg \phi \vdash \perp}{\Gamma \vdash \phi} \\
\\
(\text{I}_{\wedge}) \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \quad (\text{E}_{\wedge 1}) \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \quad (\text{E}_{\wedge 2}) \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \\
\\
(\text{I}_{\vee 1}) \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \quad (\text{I}_{\vee 2}) \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \quad (\text{E}_{\vee}) \frac{\Gamma \vdash \phi \vee \psi \quad \Gamma, \phi \vdash \theta \quad \Gamma, \psi \vdash \theta}{\Gamma \vdash \theta} \\
\\
(\text{I}_{\forall}) \frac{\Gamma \vdash \phi[y/x]}{\Gamma \vdash \forall x. \phi} \text{ (a)} \quad (\text{E}_{\forall}) \frac{\Gamma \vdash \forall x. \phi}{\Gamma \vdash \phi[t/x]} \\
\\
(\text{I}_{\exists}) \frac{\Gamma \vdash \phi[t/x]}{\Gamma \vdash \exists x. \phi} \quad (\text{E}_{\exists}) \frac{\Gamma \vdash \exists x. \phi \quad \Gamma, \phi[y/x] \vdash \theta}{\Gamma \vdash \theta} \text{ (b)}
\end{array}$$

(a) y must not occur free in Γ or ϕ
 (b) y must not occur free in Γ, ϕ or θ

Figure 3.2: System \mathcal{N}_{FOL} for classical first-order logic
[AFPMdS11]

3.1.3 Hoare Logic

In 1969, Hoare [Hoa69] introduced an axiomatic approach for reasoning about the correctness of imperative programs, based on first-order logic. A program is seen as a mathematical object and properties of a program are established through the use of an inference system.

Hoare, who was inspired by the work of Robert Floyd [Flo67], used a very simple language (known as **While language**) — that has only the following statements: assignment, sequence of statements, conditional *if-then-else*, and loop *while* — to illustrate his ideas.

“The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program.”

Robert Floyd, in [Flo67]

Although simple, the introduction of Hoare logic had a significant impact on methods for both designing and verifying programs. It owes its success mainly to two factors [dRE98]:

- It is *state-based*, by characterizing programming constructs as transformers of states, and therefore applies in principle to every such construct.
- It is *syntax directed*: every rule for a programming construct reduces the problem of proving properties of that construct to proving properties of its constituent constructs.

Hoare logic deals with the notion of correctness of a program by means of *preconditions* and *postconditions*. A program is said to be correct with respect to a precondition and a postcondition if after the execution of the program the postcondition holds, if the program was started in a state which satisfied the precondition.

Additionally, if there are loops in a program they need to be annotated with *invariants*. An invariant states what should be true on entry into a loop and what is guaranteed to remain true in every iteration of the loop. This means that on exit from the loop, the loop invariant and the loop termination condition are guaranteed to hold.

In the following subsections, we present a version of Hoare logic for annotated procedures. The reader is referred to [AFPMdS11, FP11] for more details.

Exp_{int}	$\ni e$	$::= \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid x \mid$ $-e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \text{ div } e_2 \mid e_1 \bmod e_2 \mid$ $a[e]$
Exp_{bool}	$\ni b$	$::= \top \mid \perp \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid$ $e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2$
Assert	$\ni A$	$::= \top \mid \perp \mid e = e \mid e < e \mid e \leq e \mid e > e \mid e \geq e \mid$ $e \neq e \mid A \wedge A \mid A \vee A \mid \neg A \mid A \rightarrow A \mid \forall x. A \mid \exists x. A$
Comm	$\ni C$	$::= \text{skip} \mid x := e \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } \{A\} S \mid$ $\text{call } p$
Block	$\ni S$	$::= C \mid C ; S$
Proc	$\ni \Phi$	$::= \text{pre } A \text{ post } A \text{ proc } p = S$
Prog	$\ni \Pi$	$::= \Phi \mid \Pi \Phi$

Figure 3.3: Abstract syntax of While^{DbC} language (x and p range over sets of *variables* and *procedure names* respectively).

The While^{DbC} Programming Language

In this subsection will be presented the syntax of a simple programming language, hereafter called While^{DbC} , an extension of the *While* language with mutually recursive procedures annotated with contracts.

The language has two basic types,

$$\mathbf{Type} \ni \tau ::= \mathbf{bool} \mid \mathbf{int}$$

and its syntax is given in Figure 3.3. It is defined in two levels: first, it is possible to form sequences of commands, which are standard programs of a *While* programming language. It is then possible to construct procedures consisting of a block of code, annotated with a precondition and a post-condition that forms the procedure's specification, or *contract*. Commands include **skip**, assignment, conditional branching, loops, and a procedure call command.

Each block may additionally be annotated with loop invariants, if it contain loops. The language of annotated assertions (used for writing invariants and contracts) extends boolean expressions with implication and first-order quantification.

Expressions of type **int** are interpreted as values in \mathbb{Z} . Operators of the language are interpreted as the corresponding operators in the semantic

domain: $e_1 + e_2$ is interpreted as the integer addition of e_1 and e_2 , $e_1 \leq e_2$ is interpreted as “ e_1 is less than or equal to e_2 ”, and so on.

A *program* is a non-empty set of (mutually recursive) procedure definitions. For the sake of simplicity, only *parameterless procedures* will be considered (sharing a set of global variables), but the ideas presented here can be adapted to cope with parameters (passed by value or by reference) as well as return values of functions (see Chapter 8 of [AFPMdS11]).

From now on, a program will be considered well-formed if the name of every procedure defined in it is unique and the program is closed with respect to procedure invocation. We will write $\mathbf{PN}(\Pi)$ for the set of names of procedures defined in Π . The operators **pre**, **post**, and **body** will be used to refer to a procedure’s precondition, postcondition, and body command, respectively, i.e. given the procedure definition **pre** P **post** Q **proc** $p = S$ with $p \in \mathbf{PN}(\Pi)$, one has $\mathbf{pre}_\Pi(p) = P$, $\mathbf{post}_\Pi(p) = Q$, and $\mathbf{body}_\Pi(p) = S$. The program name will be omitted when clear from context.

Hoare Logic for While^{DbC} Programs

The basic formulas of Hoare logic are *partial correctness* formulas $\{\phi\}S\{\psi\}$, also called *Hoare triples*, where ϕ and ψ are assertions.

ϕ is called a *precondition* and is composed of assertions that are assumed to hold when the execution of the program is started. ψ is called a *postcondition* and is composed of assertions that must hold when execution stops.

The pair of assertions (ϕ, ψ) is the intended *specification* for the block S . The correctness of a program is always defined with respect to a given specification.

A *partial correctness* property for a block S with respect to a specification (ϕ, ψ) has the following meaning: if ϕ holds in a given state and S is executed in that state, then either execution of S does not stop, or if it does, ψ will hold in the final state.

A *total correctness* property for a block S , written $[\phi]S[\psi]$, with respect to a specification (ϕ, ψ) has the following meaning: if ϕ holds in a given state and S is executed in that state, then execution of S will stop, and moreover ψ will hold in the final state of execution.

The validity of a Hoare triple is established by the following interpretation: $\llbracket \{\phi\}S\{\psi\} \rrbracket = \text{For all states } s, s', \llbracket P \rrbracket(s) \wedge (S, s) \Downarrow s' \Rightarrow \llbracket Q \rrbracket(s')$. Note that the annotations plays no role in the operational semantics of the language.

There are three kinds of variables that may occur in a Hoare triple:

- *Logical variables*: variables that are *bound* (recall Definition 33) by some quantifier in ϕ or ψ or in annotations in S .

- *Program variables*: variables that are used in commands of S , loop annotations, as well as in the precondition ϕ and postcondition ψ .
- *Auxiliary variables*: variables that occur free in ϕ , ψ or other annotations inside S , but do not occur as program variables in S .

Occurrences of variables in the precondition and postcondition of a block refer to their values in the pre-state and post-state of execution of the block respectively; the use of *auxiliary variables* (that occur in assertions only, not in the code) allows the value of a variable in the pre-state to be recorded, which can thus be used in the postcondition.

Proof System

The Hoare calculus is a set of logical rules (an inference system) for reasoning about Hoare triples. We will refer to it as system \mathcal{H} . It describes in a formal way the use of preconditions, postconditions, and invariants in order to verify blocks of commands and programs.

The inference system of Hoare logic is composed of the following rules:

$$(skip) \quad \frac{}{\{\phi\} \mathbf{skip} \{\phi\}}$$

The **skip** axiom states that this command preserves the truth of assertions (does not change the state). If ϕ holds before its execution, then ϕ thus holds afterwards as well.

$$(assign) \quad \frac{}{\{\psi[e/x]\} x := e \{\psi\}}$$

The *assign* axiom states that to make sure that the assertion ψ holds after the assignment of e to x , it suffices to make sure that the assertion resulting from substituting e for x in ψ holds in the pre-state.

$$(seq) \quad \frac{\{\phi\} C \{\theta\} \quad \{\theta\} S \{\psi\}}{\{\phi\} C ; S \{\psi\}}$$

The *seq* rule introduces an intermediate assertion and states that to prove that ϕ holds before the execution of $C ; S$ then ψ holds afterwards, it suffices to show for some θ that: if ϕ holds before C is executed then θ holds afterwards and if θ holds before S is executed then ψ holds afterwards.

$$(while) \quad \frac{\{\theta \wedge b\} S \{\theta\}}{\{\theta\} \mathbf{while} \ b \ \mathbf{do} \ \{\theta\} S \{\theta \wedge \neg b\}}$$

The *while* rule states that:

- In order to prove that the assertion θ is a loop invariant, then it is required to prove that θ is preserved by the body of the loop when the loop condition also holds as a precondition.

- The negation of the loop condition is a postcondition of the loop.

$$(if) \frac{\{\phi \wedge b\} S_t \{\psi\} \quad \{\phi \wedge \neg b\} S_f \{\psi\}}{\{\phi\} \text{ if } b \text{ then } S_t \text{ else } S_f \{\psi\}}$$

The *if* rule states that in order to show that ϕ holds before the execution of the conditional then ψ holds, in the final state it suffices to show that the same is true for each condition branch, under the additional assumption that this branch is executed (this is in accordance with the fact that S_t is executed when b holds and S_f when $\neg b$ holds).

The next two rules make use of a new judgement $\{\Pi\}$, whose interpretation is that every procedure in the program respects its contracts, i.e. it is correct with respect to its annotated precondition and postcondition.

(mutual recursion – parameterless)

$$\frac{\begin{array}{c} \{\Pi\} \\ \vdots \\ \{\mathbf{pre}_\Pi(\mathbf{p}_1)\} \mathbf{body}_\Pi(\mathbf{p}_1) \{\mathbf{post}_\Pi(\mathbf{p}_1)\} \end{array} \quad \cdots \quad \begin{array}{c} \{\Pi\} \\ \vdots \\ \{\mathbf{pre}_\Pi(\mathbf{p}_n)\} \mathbf{body}_\Pi(\mathbf{p}_n) \{\mathbf{post}_\Pi(\mathbf{p}_n)\} \end{array}}{\{\Pi\}}$$

The *mutual recursion-parameterless* rule states that a program is correct if each one of its individual procedures is correct. If the correctness of the body of each procedure in the program can be derived assuming only the correctness of each procedure in that same program, then all the procedures (and therefore the program) are correct.

(procedure call – parameterless)

$$\frac{\{\Pi\}}{\{\phi\} \mathbf{call} \mathbf{p} \{\psi\}} \quad \text{if} \quad \phi \rightarrow \forall \bar{x}_f. (\forall \bar{y}_f. \mathbf{pre}(\mathbf{p})[\bar{y}_f/\bar{y}] \rightarrow \mathbf{post}(\mathbf{p})[\bar{y}_f/\bar{y}, \bar{x}_f/\bar{x}] \rightarrow \psi[\bar{x}_f/\bar{x}])$$

where

$p \in \mathbf{PN}(\Pi)$

\bar{y} is a sequence of the auxiliary variables of \mathbf{p}

\bar{x} is a sequence of the program variables occurring in $\mathbf{body}(\mathbf{p})$

\bar{x}_f and \bar{y}_f are sequences of fresh variables

The expression $t[\bar{e}/\bar{x}]$, with $\bar{x} = x_1, \dots, x_n$ and $\bar{e} = e_1, \dots, e_n$, denotes the parallel substitution $t[e_1/x_1, \dots, e_n/x_n]$

The *procedure call-parameterless* rule concerns the procedure call command, and has as premise the global correctness of the program. The side

condition establishes the adaptation between the actual required specification and the contract [Kle99].

$$(conseq) \frac{\{\phi\} C \{\psi\}}{\{\phi'\} C \{\psi'\}} \quad \text{if } \phi' \rightarrow \phi \text{ and } \psi \rightarrow \psi'$$

The *conseq* rule has a side condition stating that the first-order formulas must be valid. This rule can be applied to infer a triple from another triple containing the same program, in which either the precondition has been weakened or the postcondition has been strengthened. An immediate consequence of this rule is that derivations for a given goal are not unique.

Proof System for Total Correctness

The validity of the Hoare triple $\{\theta \wedge b\} S \{\theta\}$ implies the validity of

$$\{\theta\} \textbf{while } b \textbf{ do } \{\theta\} S \{\theta\}.$$

However, the validity of $[\theta \wedge b] S [\theta]$ does not imply the validity of

$$[\theta] \textbf{while } b \textbf{ do } \{\theta\} S [\theta].$$

In order to prove total correctness one must prove the termination of the loop. In this situation, loop invariants are inappropriate since they do not include any information about loop termination. To deal with this a mathematical function is required, which is restricted to non-negative numbers and whose value is monotonically decreasing in each loop iteration. This function is usually called a *loop variant*. The presence of a loop variant implies the termination of all possible executions of the loop.

In a total correctness setting the Hoare logic rule for while loops becomes:

$$\frac{[\theta \wedge b \wedge V = v_0] C [\theta \wedge V < v_0]}{[\theta] \textbf{while } V \textbf{ do } \{\theta, V\} b [\theta \wedge \neg b]} \quad \text{if } \theta \wedge b \rightarrow V \geq 0$$

where V is the loop variant and v_0 is a fresh auxiliary variable used to record its initial value.

All the remaining rules of the total correctness system are similar to the rules of the partial correctness system.

3.1.4 Other Definitions

We end this section with some basic structures used in the context of formal verification.

Definition 45 (Boolean Function). *A Boolean Function is a function of the form $f : \mathbb{B}^k \rightarrow \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ is a boolean domain and k is a non-negative integer called the arity of the function.*

Every k -ary Boolean function can be expressed as a propositional formula in k variables x_1, x_2, \dots, x_k .

Definition 46 (Binary Decision Tree). *A Binary Decision Tree (BDT, for short) is a data structure (a tree) used to represent a Boolean function. If d is a BDT, then:*

1. *the internal nodes of d are labeled with variables;*
2. *the leaves of d are labeled with 0 and 1;*
3. *every internal node in d has exactly two children, and the two edges from d to the children are labeled with 0 (shown as a dashed line, also known as low child) and by 1 (shown as a solid line, also known as high child);*
4. *nodes in every path in d have unique labels, i.e., every two different nodes in the path are labeled with distinct variables.*

For example, Figure 3.4 and Table 3.1, show respectively a binary decision tree and a truth table representing the following Boolean function:

$$f(x_1, x_2, x_3) = (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2)$$

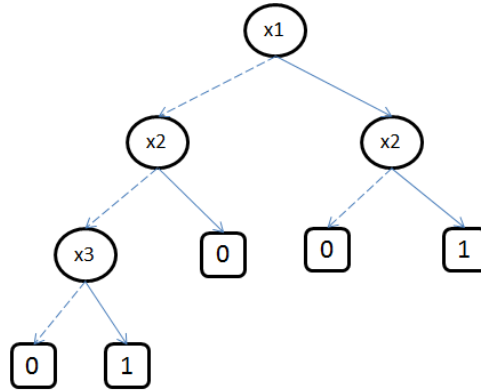


Figure 3.4: Binary Decision Tree for the function $f(x_1, x_2, x_3)$.

Therefore, to find the value of $f(x_1 = 0, x_2 = 0, x_3 = 1)$, begin at x_1 , traverse down two dashed lines to x_3 (since x_1 and x_2 are assigned to zero), then down one solid line (since x_3 is assigned to one). This leads to the terminal 1, which is the value of $f(0, 0, 1)$.

Notice the following two properties of Binary Decision Trees:

- In general, the size of a binary decision tree for a formula φ is exponential in the size of φ .

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 3.1: Truth table for the boolean function $f(x_1, x_2, x_3)$.

- Satisfiability and validity checking on binary decision trees can be done in linear time in the size of the tree.

Indeed, given the binary decision tree for a formula φ , we can verify if φ is satisfiable by simply inspecting all the leaf nodes of the tree: φ is satisfiable if and only if at least one of them is 1. Likewise, φ is valid if and only if all leaves in φ are 1.

Definition 47 (Binary Decision Diagram). *A Binary Decision Diagram (BDD, for short) is a rooted Directed Acyclic Graph (DAG) d such that d satisfies the properties of Definition 46 and additionally the following two requisites:*

- *For every node n , its left and right sub-DAGs are distinct;*
- *Every pair of sub-DAGs of d rooted at two different nodes n_1, n_2 are non-isomorphic.*

These two conditions formalize the properties that d contains no redundant tests and that the isomorphic sub-DAGs of d are merged.

As an example, the binary decision tree of Figure 3.4 can be transformed into a *binary decision diagram* as shown in Figure 3.5.

There exists a two-step algorithm to transform a Binary Decision Tree into a Binary Decision Diagram. Let $neg(n)$ denote the subtree rooted at the dashed edge coming from n and $pos(n)$ denote the subtree rooted at the solid edge. Then:

- *Elimination of Redundant Tests:* if there exists a node n such $neg(n)$ and $pos(n)$ are the same DAG, then remove this node, i.e., replace the sub-DAG rooted at n by $neg(n)$.
- *Merging isomorphic sub-DAGs:* if the sub-DAGs rooted at two different nodes n_1 and n_2 are isomorphic, then merge them into one, i.e., remove the sub-DAG rooted at n_2 and replace all edges to n_2 by edges leading to n_1 .

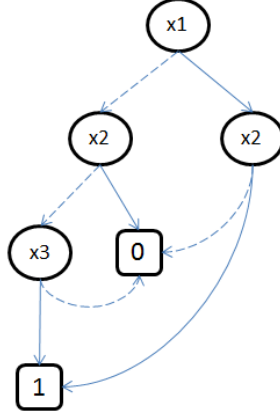


Figure 3.5: Binary Decision Diagram for the function $f(x_1, x_2, x_3)$ with $x_1 = 0$, $x_2 = 0$ and $x_3 = 1$.

Binary Decision Diagrams enjoy the following properties:

- In general, the size of a binary decision diagram for a formula φ is exponential in the size of φ .
- Satisfiability and validity checking on binary decision diagrams can be done in *constant* time.

Thus, BDDs have advantages over binary decision trees in terms of efficiency.

Finally, we briefly review two notions of transition systems augmented with labeling functions, and extensively used in model checking [BBF⁺01, BK08]. The first includes a labeling of states, and the second a labeling of transitions.

Definition 48 (Kripke Structure). *Let AP be a set of atomic propositions. A Kripke structure over AP is a structure $M = (S, S_0, R, L)$, where:*

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a total binary relation on S , so for every $s \in S$, there exists a $t \in S$, such that sRt . R represents the set of transitions of M .
- $L : S \rightarrow \mathcal{P}(AP)$ is a function that assigns the set of atomic propositions that hold in a state.

A path π in M is an infinite sequence s_0, s_1, s_2, \dots , such that $\forall i, s_i \in S \wedge s_i R s_{i+1}$.

Figure 3.6 shows a Kripke structure over AP , where $M = (S, S_0, R, L)$ is as follows:

- $AP = \{p, q\}$
- $S = \{s_0, s_1, s_2, s_3\}$
- $S_0 = \{s_0\}$
- $R = \{(s_0, s_1), (s_1, s_0), (s_1, s_3), (s_3, s_3), (s_0, s_2), (s_2, s_1)\}$
- $L(s_0) = \{\}, L(s_1) = \{p, q\}, L(s_2) = \{q\}, L(s_3) = \{p\}$

Notice that without the loop (s_3, s_3) , R would not be total, and M would not be a Kripke structure.

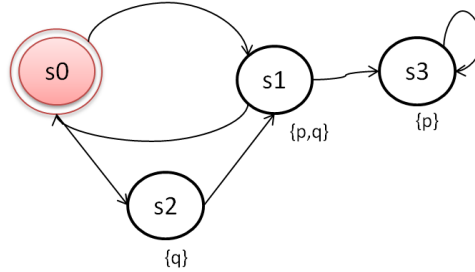


Figure 3.6: Kripke Structure: Example

Definition 49 (Labeled Transition System). *A transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta)$ is given by a set S of states, a nonempty subset $I \subseteq S$ of initial states, a set \mathcal{A} of actions, and a transition relation $\delta \subseteq S \times \mathcal{A} \times S$.*

An action $A \in \mathcal{A}$ is called enabled at state $s \in S$ iff $(s, A, t) \in \delta$ holds for some $t \in S$.

A run of a Labeled Transition System is a list of transitions that proceed from one state to another.

The trace of a run is the series of labels from these transitions.

Both runs and traces may be finite or infinite.

A labeled transition system specifies the allowed evolutions of the system: starting from an initial state, the system evolves by performing actions that take the system to new states.

Figure 3.7 shows a Labeled Transition System over T_1 , where $T_1 = (S, I, \mathcal{A}, \delta)$ is as follows:

- $S = \{In, R, A\}$
- $I = \{In\}$
- $\mathcal{A} = \{\text{alert, relax, on, off}\}$

- $\delta = \{ (In, \text{alert}, R), (R, \text{relax}, In), (R, \text{on}, A), (A, \text{off}, In) \}$

The following are examples of traces over T_1 :

on.off, alert.on.off, alert.relax.alert.on.off.allert.

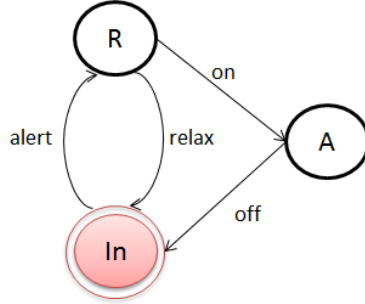


Figure 3.7: Labeled Transition System: Example

3.2 Static Techniques: Semi-automated

After this not so short introduction of basic concepts, let us return to the main topic of this chapter — Program Verification — starting more precisely with static approaches (semi-automated in this section and fully automated in the next one).

When program verification first appeared in the 60's with Floyd [Flo67], Hoare [Hoa69] and Dijkstra [Dij76b], the primary focus was on manual reasoning. Hoare's approach was to regard source code as specifying a relation between assertions.

However, as the size of software systems grew, the use of manual techniques became error-prone as well as too cumbersome. Besides these problems, questions related with the correctness of long and arduous proofs were raised. Mechanical proofs (constructed manually with the assistance of a computer program) are safer, but even more detailed and difficult to understand. Over the years, a movement towards semi-automatic techniques (through the use of loop invariants and pre/post conditions) appeared. They are called *semi-automatic* due to the need for the programmer to include assertions in the program in order to prove its correctness. The presence of annotations (in particular loop invariants) eliminates the difficulty inherent to the rules of Hoare logic. One solution to deal with the difficulty of proving the correctness of a program is to *partially* automate the construction of a proof. This is usually done by what is called *Verification Condition Generator* (VCGen for short). A VCGen reduces the question of whether a program is consistent with its specification to that of whether certain logical formulas are theorems in an underlying theory. These sets of formulas are

usually called *verification conditions* and they do not contain any references to programming language constructs. They only deal with the logic of the underlying data types. The provability of these theorems is sufficient to guarantee that an axiomatic proof using the Hoare system could be constructed. A VCGen must thus embody the semantics of the programming language; they can also be seen as implementations of Hoare-style axiomatic semantics. Note that even through the inclusion of annotations makes possible to generate verification conditions mechanically, this does not mean that proofs can always be done automatically, since they are still restricted by the decidability properties of the underlying logic.

In the past, all VCGens have been hand-coded for a specific language. Nowadays, the scenario is different, since a number of VCGens are available based on *intermediate languages*; the idea being that is that different programming languages can be translated into such intermediate language. Also, nowadays checking verification conditions is frequently left to an external theorem prover or solver, making the process more effective. Figure 3.8 depicts the general architecture of a program verifier that makes use of a VCGen.

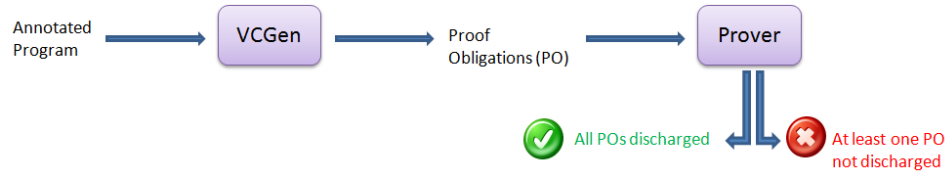


Figure 3.8: General architecture of a program verifier using a VCGen

From an engineer’s perspective, program verifiers are similar to interpreters. Their input is a program written in a high-level language, and the output is a set of warnings or errors. Using a good program verifier, the lack of warnings and errors should be an indicator that the source program is correct.

Before proceeding to a more detailed discussion of VC generation, it is worth mentioning Dijkstra’s related *predicate transformers*:

- One that maps a program fragment S and a postcondition Q into a precondition. This function computes an assertion *sufficient* to guarantee that Q will be obtained as a postcondition. A predicate transformer that produces preconditions which are both *necessary* and *sufficient* to derive Q as the postcondition is said to compute the *weakest derivable precondition* and is usually denoted by $\text{wp}.S.Q$.

The provability of the verification condition $P \rightarrow \text{wp}.S.Q$ in the underlying logic is thus sufficient to show that $[P]S[Q]$ is derivable within any Hoare system containing the rule of consequence.

The weakest precondition calculus was introduced by Dijkstra in his paper “Guarded commands, nondeterminacy and formal derivation of programs” [Dij75], in which the semantics of an imperative language is defined by assigning to each statement a corresponding predicate transformer.

Dijkstra’s *wp* predicate transformer implies the termination of the program. The related *weakest liberal precondition* (*wlp*) predicate transformer, which does not force termination, is appropriate for establishing the derivability of partial correctness triples.

- Another approach, dual of the previous one, maps a program fragment S and a precondition P into a postcondition. This predicate is usually known as *strongest postcondition* calculus and denoted by $\text{sp}.S.P$.

$\text{sp}.S.P$ is a postcondition for S that is ensured by precondition P and thus must satisfy $\{P\}S\{\text{sp}.S.P\}$. Taking any P, Q such that $\{P\}S\{Q\}$, then $\text{sp}.S.P \rightarrow Q$.

As a consequence, a Hoare triple $\{P\}S\{Q\}$ is provable in Hoare logic if and only if the following predicate holds: $\text{sp}.S.P \rightarrow Q$.

Notions of weakest preconditions and strongest postconditions for annotated blocks of code will be defined in what follows, and play an important role in the mechanization of Hoare logic. An alternative approach is the direct use of predicate transformers after translating programs to a guarded commands language. This approach is followed very successfully in the ES-C/Java2 and Boogie projects and tools.

The next subsections will be dedicated to the presentation of:

- a) how Hoare logic can be given as alternative formulation in order to deal with the backward application of rules of the logic without ambiguity (this ambiguity was introduced by the *seq/conseq* rules as explained in subsection 3.1.3);
- b) the definition of weakest preconditions for one language;
- c) a VCGen algorithm based on a weakest precondition strategy;
- d) the definition of strongest postconditions for one language;
- e) a VCGen algorithm based on a strongest postcondition strategy.

Some of these and other related issues will be discussed in more detail in Section 6.1, in particular verification conditions for total correctness and a VCGen containing the use of strongest postconditions and weakest preconditions.

Some tools that implement VCGens are presented in Section 3.5.

3.2.1 Mechanizing Hoare Logic

There are two desirable properties that the Hoare calculus should enjoy in order to be used as the underlying logic of a VCGen [AFPMdS11]:

- All the assertions that occur in the premises of a rule also occur in its conclusion — this property avoids the need to *guess* assertions that appear in the new goals generated when a rule is applied. The exceptions of the inference system of Hoare logic are (recall the rules from subsection 3.1.3): the *seq* rule where an intermediate assertion needs to be guessed; and the *conseq* rule where both a precondition and a postcondition must be guessed. Loop invariants do not have to be invented because they are part of the program to start with.
- The absence of ambiguity in the choice of a rule. When applying a rule, it would be desirable to have a single rule that could be applied to a given goal. The *conseq* rule introduces this ambiguity too, as it can be applied with any arbitrary goal.

The inference system presented in subsection 3.1.3 thus needs to be adapted in order to allow a mechanization of Hoare logic. Figure 3.9 presents such an alternative. In [AFPMdS11], it is proved that systems \mathcal{H} and \mathcal{H}_g are equivalent.

By eliminating the *conseq* rule, ambiguity was also eliminated. Notice, however, that the *seq* rule is still present, which means that certain intermediate assertions still have to be guessed and thus it is possible to construct different proof trees (with the same structure) for a given Hoare triple. This problem can be solved by using a strategy (based, for instance, on weakest preconditions) as a way to systematically determine the intermediate assertions required by the *seq* rule.

3.2.2 The Weakest Precondition Strategy

As referred in the previous subsection, the presence of the *seq* rule introduces some ambiguity due to the need to guess an intermediate assertion. The general strategy to deal with this problem is as follows. Consider that a derivation is being constructed for the Hoare triple $\{\phi\}C\{\psi\}$, where ϕ may be either known or unknown (if unknown the triple will be written as $\{?\}C\{\psi\}$).

If ϕ is known, then the derivation is constructed by applying the unique rule applicable from Figure 3.9. In case the goal is of the form $\{\phi\}C_1; C_2\{\psi\}$, then the second sub-derivation will be first constructed with a goal of the form $\{?\}C_2\{\psi\}$. After the construction of this sub-derivation $?$ will be instantiated with some assertion θ , and then the first sub-derivation can be constructed for the goal $\{\phi\}C_1\{\theta\}$.

$$\begin{array}{l}
(\textit{skip}) \quad \frac{}{\{\phi\} \mathbf{skip} \{\psi\}} \quad \text{if } \phi \rightarrow \psi \\
\\
(\textit{assign}) \quad \frac{}{\{\phi\} x := e \{\psi\}} \quad \text{if } \phi \rightarrow \psi[e/x] \\
\\
(\textit{seq}) \quad \frac{\{\phi\} C_1 \{\theta\} \quad \{\theta\} C_2 \{\psi\}}{\{\phi\} C_1 ; C_2 \{\psi\}} \\
\\
(\textit{while}) \quad \frac{\{\theta \wedge b\} C \{\theta\}}{\{\phi\} \mathbf{while } b \mathbf{ do } \{\theta\} C \{\psi\}} \quad \text{if } \phi \rightarrow \theta \text{ and } \theta \wedge \neg b \rightarrow \psi \\
\\
(\textit{if}) \quad \frac{\{\phi \wedge b\} C_t \{\psi\} \quad \{\phi \wedge \neg b\} C_f \{\psi\}}{\{\phi\} \mathbf{if } b \mathbf{ then } C_t \mathbf{ else } C_f \{\psi\}} \\
\\
(\textit{mutual recursion} - \textit{parameterless}) \quad \frac{\begin{array}{c} [\{\Pi\}] \\ \vdots \\ \{\mathbf{pre}_\Pi(\mathbf{p}_1)\} \mathbf{body}_\Pi(\mathbf{p}_1) \{\mathbf{post}_\Pi(\mathbf{p}_1)\} \end{array} \quad \cdots \quad \begin{array}{c} [\{\Pi\}] \\ \vdots \\ \{\mathbf{pre}_\Pi(\mathbf{p}_n)\} \mathbf{body}_\Pi(\mathbf{p}_n) \{\mathbf{post}_\Pi(\mathbf{p}_n)\} \end{array}}{\{\Pi\}} \\
\\
(\textit{procedure call} - \textit{parameterless}) \quad \frac{\{\Pi\}}{\{\phi\} \mathbf{call } \mathbf{p} \{\psi\}} \quad \text{if } \phi \rightarrow \forall \overline{x_f}. (\forall \overline{y_f}. \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}] \rightarrow \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}] \rightarrow \psi[\overline{x_f}/\overline{x}])
\end{array}$$

Figure 3.9: Inference system of Hoare logic without consequence rule: system \mathcal{H}_g

If ϕ is unknown, the construction follows as explained in the previous item, except that, in the rules for **skip**, assignments and loop, with a side condition $\phi \rightarrow \theta$ for some θ , the precondition ϕ is taken to be θ .

This strategy ensures that a weakest precondition is determined for goals that have unknown preconditions: a derivation for the Hoare triple $\{\phi\}C_1; C_2\{\psi\}$ is constructed by taking as postcondition for C_1 the weakest precondition θ that will ensure the validity of ψ after execution of C_2 .

Before formalizing a VCGen based on this strategy, let us consider how weakest preconditions should be defined for one language annotated with invariants.

Figure 3.10 shows the rules for computing the weakest precondition for a given program S and a postcondition Q . Notice that:

- The **wprec** function, in the case of the sequence rule, is invoked with argument **wprec**(S, ψ), to calculate the postcondition for the first command in the sequence.
- **wprec**($x := e, \psi$) = $\psi[e/x]$ denotes the substitution of e for x in ψ . For example, the **wprec**($x := x - 3, x > 15$) is $x - 3 > 15 \equiv x > 18$.
- The weakest precondition of a conditional command is obtained from the weakest preconditions of both branches.
- The weakest precondition of a loop **while** b **do** $\{\theta\} C$ is simply defined as being its invariant θ , since it is not possible to derive triples of the form

$$\{\phi\} \text{while } b \text{ do } \{\theta\} C \{\psi\}$$

such that ϕ is weaker than θ .

This approach differs from the Dijkstra's approach. The definition of **wprec** shares much with Dijkstra's *weakest liberal precondition* predicate transformer; the difference is that whereas a true weakest precondition is given as a least-fixpoint solution to a recursive equation, **wprec** simply makes use of the annotated loop invariant. Note that it may well be the case that this annotation is not in fact an invariant; even if it is an invariant, it is possible that it is not sufficiently strong to allow Q as a postcondition of the loop; on the other hand, the annotation does not need to be the weakest of all sufficiently strong invariants, and often is not. Thus, the function that computes the weakest precondition of a command will be denoted as **wprec** instead of simply **wp**. To maintain the consistency, the function that computes the strongest postcondition of a command will be denoted as **spost**. **wprec** computes a weak precondition of an annotated program, which is not necessarily the weakest precondition of the underlying (non-annotated) program.

$$\begin{aligned}
\text{wprec}(\text{skip}, \psi) &= \psi \\
\text{wprec}(x := e, \psi) &= \psi[e/x] \\
\text{wprec}(C; S, \psi) &= \text{wprec}(C, \text{wprec}(S, \psi)) \\
\text{wprec}(\text{if } b \text{ then } S_t \text{ else } S_f, \psi) &= (b \rightarrow \text{wprec}(S_t, \psi)) \wedge (\neg b \rightarrow \text{wprec}(S_f, \psi)) \\
\text{wprec}(\text{while } b \text{ do } \{\theta\} S, \psi) &= \theta \\
\text{wprec}(\text{call } \mathbf{p}, \psi) &= \forall \bar{x}_f. (\forall \bar{y}_f. \mathbf{pre}(\mathbf{p})[\bar{y}_f/\bar{y}] \\
&\quad \rightarrow \mathbf{post}(\mathbf{p})[\bar{y}_f/\bar{y}, \bar{x}_f/\bar{x}]) \rightarrow \psi[\bar{x}_f/\bar{x}]
\end{aligned}$$

where \bar{y} is a sequence of the auxiliary variables of \mathbf{p}
 \bar{x} is a sequence of the program variables occurring in $\mathbf{body}(\mathbf{p})$
 \bar{x}_f and \bar{y}_f are sequences of fresh variables
 The expression $t[\bar{e}/\bar{x}]$, with $\bar{x} = x_1, \dots, x_n$ and $\bar{e} = e_1, \dots, e_n$,
 denotes the parallel substitution $t[e_1/x_1, \dots, e_n/x_n]$

Figure 3.10: Weakest precondition of annotated blocks

3.2.3 A VCGen Algorithm based on Weakest Preconditions

This section introduces a VCGen algorithm based on the weakest precondition strategy. Basically, for a given Hoare triple $\{\phi\}S\{\psi\}$, a weakest precondition $\text{wprec}(S, \psi)$ is calculated which is required for ψ to hold after terminating the execution of S .

A first approach would be simply to ensure that the precondition ϕ is stronger than the calculated weakest precondition. The VCGen would simply be defined as:

$$\text{VCG}^w(\phi, S, \psi) = \{\phi \rightarrow \text{wprec}(S, \psi)\}$$

However, this definition would only work if the program does not contain loops. Loops introduce additional side conditions that need to be recursively collected. According to the weakest precondition strategy for the construction of proof trees, whenever a precondition ϕ occurs in a side condition of the form $\phi \rightarrow \psi$, the strategy determines that ϕ is taken to be equal to ψ (the weakest precondition), and so this particular verification condition is trivially satisfied.

Thus, the rule for while loops introduces two verification conditions:

- The first one corresponds to the preservation of the loop invariant. Note that the loop rule in system \mathcal{H}_g has as premise a Hoare triple with a different precondition from the one in the conclusion. This forces the

$$\begin{aligned}
\text{VC}^w(\text{skip}, \psi) &= \emptyset \\
\text{VC}^w(x := e, \psi) &= \emptyset \\
\text{VC}^w(C; S, \psi) &= \text{VC}^w(C, \text{wprec}(S, \psi)) \cup \text{VC}^w(S, \psi) \\
\text{VC}^w(\text{if } b \text{ then } C_t \text{ else } C_f, \psi) &= \text{VC}^w(C_t, \psi) \cup \text{VC}^w(C_f, \psi) \\
\text{VC}^w(\text{while } b \text{ do } \{\theta\} S, \psi) &= \{\theta \wedge b \rightarrow \text{wprec}(S, \theta), \theta \wedge \neg b \rightarrow \psi\} \cup \text{VC}^w(S, \theta) \\
\text{VC}^w(\text{call } p, \psi) &= \emptyset
\end{aligned}$$

Figure 3.11: VC^w auxiliary function

inclusion of a verification condition stating that the intended precondition $\theta \wedge b$ must be stronger than the calculated weakest precondition of the loop body, with respect to the invariant θ .

- The second one corresponds to the second side condition of the while rule (the first side condition is trivially satisfied by the use of weakest preconditions).

The VCGen algorithm for blocks containing loops can now be defined:

Definition 50 (Verification Conditions for a block based on weakest preconditions).

$$\text{VCG}^w(\phi, S, \psi) = \{\phi \rightarrow \text{wprec}(S, \psi)\} \cup \text{VC}^w(S, \psi)$$

Where the auxiliary function VC^w is defined in Figure 3.11.

Now that the set of verification conditions for a command block has been defined, the algorithm to compute the set of verification conditions for a program can be given. The VCs for a program can be defined as the union of all the sets of verification conditions required for its constituent procedures.

Definition 51 (Verification Conditions for a Program based on weakest preconditions).

$$\text{Verif}(\{\Pi\}) = \bigcup_{p \in \text{PN}(\Pi)} \text{VCG}^w(\text{pre}_\Pi(p), \text{body}_\Pi(p), \text{post}_\Pi(p))$$

Given a procedure p , $\text{VCG}^w(\text{pre}(p), \text{body}(p), \text{post}(p))$ will in this thesis be called its set of *local verification conditions*.

The intra-procedural (command-level) aspects of the VCGen are standard, but the inter-procedural aspects (program-level) are less well-known. We remark the following:

- Although this is somewhat hidden (unlike in the underlying program logic), the VCGen is based on a *mutual recursion* principle, i.e. the proof of correctness of each routine assumes the correctness of all the routines in the program, including itself. If all verification conditions are valid, correctness is established *simultaneously* for the entire set of procedures in the program. This is the fundamental principle behind *design-by-contract*.
- The weakest precondition rule for procedure call takes care of what is usually known as the *adaptation* between the procedure's contract and the postcondition required in the present context. The difficulty of reasoning about procedure calls has to do with the need to refer, in a contract's postcondition, to the values that some variables had in the pre-state. This issue will be discussed again in Section 6.1. The reader is referred to [Kle99] for details and a historical discussion of approaches to adaption.
- This VCGen is *sound*: it can be proved that a Hoare logic tree with conclusion $\{P\} S \{Q\}$ can be constructed that has exactly the assertions in the set $\text{VCG}^w(P, S, Q)$ as side conditions; if these conditions are all valid then the tree is indeed a proof tree, and the triple is derivable in Hoare logic.

3.2.4 The Strongest Postcondition Strategy

As previously mentioned, the strongest postcondition is a predicate transformer that propagates the precondition, in a forward manner, through the program.

Figure 3.12 shows the rules to compute the strongest postcondition of a sequence of statements, in one context of annotated programs with procedure calls. $\text{spost}(P, S)$ characterizes the set of all states in which there exists a computation of S that begins with P true. That is, given that P holds, execution of S results in $\text{spost}(P, S)$ being satisfied in the final state, if S terminates ($\text{spost}(P, S)$ assumes partial correctness).

Notice that, dually to the wprec function, in the case of the sequence rule, the spost function is invoked with the first command ($\text{spost}(C, P)$) to calculate the precondition for the second command (S) in the sequence.

A dual method to generate verification conditions is based on forward propagation of assertions using spost . The algorithm to verify a procedure using the strongest postcondition strategy is given as follows:

Definition 52 (Verification Conditions for a block based on strongest postconditions).

$$\text{VCG}^s(P, S, Q) = \text{VC}^s(S, P) \cup \{\text{spost}(S, P) \rightarrow Q\}$$

Where the auxiliary function VC^s is defined in Figure 3.13.

$$\begin{aligned}
\text{spost}(\mathbf{skip}, \phi) &= \phi \\
\text{spost}(x := e, \phi) &= \exists v. \phi[v/x] \wedge x = e[v/x] \\
\text{spost}(C; S, \phi) &= \text{spost}(S, \text{spost}(C, \phi)) \\
\text{spost}(\mathbf{if } b \mathbf{ then } S_t \mathbf{ else } S_f, \phi) &= \text{spost}(S_t, b \wedge \phi) \vee \text{spost}(S_f, \neg b \wedge \phi) \\
\text{spost}(\mathbf{while } b \mathbf{ do } \{\theta\} S, \phi) &= \theta \wedge \neg b \\
\text{spost}(\mathbf{call } \mathbf{p}, \phi) &= \exists \overline{x_f}. \phi[\overline{x_f}/\overline{x}] \wedge (\forall \overline{y_f}. \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}] \\
&\quad \rightarrow \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}])
\end{aligned}$$

Figure 3.12: Strongest postcondition of annotated blocks

3.3 Static Techniques: Fully-Automated

This section is dedicated to the fully automated techniques currently used to verify programs.

3.3.1 Abstract Interpretation

As discussed in Chapter 2, static analyzes compute in an *efficient* and *sound* way information about a program — for instance control flow, data dependencies and call flow — without executing it. Although the primary goal of static analysis was optimization, nowadays it is widely used in program verification.

Two types of static analysis can be distinguished:

- *Concrete interpretation*: the analysis of a program is done according to a *concrete domain*.

The static analysis approaches classified in Section 2.2, fall in this category of concrete interpretation: flow-sensitive (if the order of execution of statements is taken into account); path-sensitive (if it distinguishes between paths through a program and attempts to consider only the feasible ones); context-sensitive (if methods are analyzed based on the call sites); and inter-procedural (if a method's body is analyzed considering the context of each respective call site).

- *Abstract interpretation*: the analysis of a program is done according to an *abstract domain*. An abstract domain is an approximate representation of sets of concrete values. An abstraction function is used to map concrete values to abstract ones.

$$\begin{aligned}
VC^s(\mathbf{skip}, \phi) &= \emptyset \\
VC^s(x := e, \phi) &= \emptyset \\
VC^s(C; S, \phi) &= VC^s(C, \phi) \cup VC^s(S, \mathbf{spost}(C, \phi)) \\
VC^s(\mathbf{if } b \mathbf{ then } S_t \mathbf{ else } S_f, \phi) &= VC^s(S_t, b \wedge \phi) \cup VC^s(S_f, \neg b \wedge \phi) \\
VC^s(\mathbf{while } b \mathbf{ do } \{\theta\} S, \phi) &= \{\phi \rightarrow \theta, \mathbf{spost}(S, \theta \wedge b) \rightarrow \theta\} \cup VC^s(S, \theta \wedge b) \\
VC^s(\mathbf{call } p, \phi) &= \emptyset
\end{aligned}$$

Figure 3.13: VC^s auxiliary function

Abstract interpretation can be defined as the process of evaluating the behavior of a program on an abstract domain to obtain a rough solution [JM09]. Such interpretations can be derived from a concrete interpretation by defining counterparts of concrete operations, such as addition or union, in the abstract domain. *Cousot and Cousot* [CC79] have shown that if there are certain constraints between the abstract and concrete domains, fixed points computed in an abstract domain are guaranteed to be sound approximations of concrete fixed points.

Abstract static analysis is used in different tools to detect different types of errors. A non extensive list is shown below:

- *CodeSonar* [Gra08a], from GramaTech (already referred in chapter 2), uses inter-procedural analysis to check for buffer overflows, memory leaks, redundant loops and branch conditions in C/C++ code.
- *K7* [Ins09], from KlockWork has similar features to the previous tool and supports Java.
- *Software Architect* [PJNR⁺98] was used to identify the error leading to the failure of the Ariane 5 rocket.
- *Astrée Static Analyzer* [BCC⁺02] is a tool that uses abstract domains for finding buffer overflows and undefined results in numerical operations. This tool was used to verify the Airbus flight control software.
- *C Global Surveyor* [NAS09] (CGS), from NASA, is a static analyzer specially developed for space mission software. It analyzes a C program to find runtime errors. CGS analyzes each instruction in a program to detect the occurrence of malfunctions like the access to a

non-initialized variable, a dereference of a null pointer or out-of-bound array accesses.

3.3.2 Model Checking

Model Checking, introduced by *Clarke and Emerson* [CE82] and by *Queille and Sifakis* [QS82], is a set of algorithmic analysis techniques used in the automatic analysis of systems that have been extensively used to find subtle errors, in particular in the design of safety-critical systems. These techniques have been proven to be cost-effective and they also integrate well with conventional design methods. These advantages have contributed to the adoption of Model Checking as a standard technique for the quality assurance of systems.

The difference between Model Checking and Static Analysis is primarily historical. Static analysis methods were used to collect facts about programs by analyzing their source code and extracting information. In contrast, model checking was conceived to check possibly complex and temporal logic properties of manually constructed finite-state models. However, Steffen [Ste91] and Schmidt [Sch98] have shown, from a theoretical point of view, that static analyzers can be cast as model checking algorithms and the other way around. In practice, static analyzers and model checkers differ in their capabilities and applicability. The main difference between the two approaches lies in the fact that Model Checking is tuned to deal with reactive systems (systems that continuously interact with their environment) while static analysis is more general. Nonetheless, modern static analyzers support specification mechanisms, and software model checkers use abstraction and operate on program code, so the distinction from the practical point of view may cease to be meaningful.

The main goal of model checking is to prove that a model of a system satisfies a correctness specification [CE82]. A model is usually defined by a set of states and transitions. A *state* is defined by: a value for the program counter; the values of all variables in a program; and the configurations of the stack and the heap. A *transition* describes how the program evolves from one state to another.

The input to a model checker is thus a description of the system and a set of properties (logical formulas usually expressed in temporal logic) that are expected to hold for the system. In practice, reactive systems are described using modelling languages, including pseudo programming languages such as **Promela**² (**Process Meta Language**). The operational semantics for these formalisms is defined in terms of transition systems. The model checking algorithm will exhaustively search for the reachable states of the system. If a state violates one of the properties, an execution trace (or run)

²<http://spinroot.com/spin/Man/Quick.html>

demonstrating the error should be produced. Such a run is usually called a *counter-example*, and can provide valuable feedback and also point to design errors.

Usually, a model checker verifies *safety* and *liveness* properties [Lam77]. *Safety* properties express that something bad never happens. These properties include assertion violations, null pointer dereferences, buffer overflow, API contract violations (order of function calls not satisfied), and so on. *Liveness* properties express that something good will happen eventually. Examples of liveness properties are: “If the tank is empty, the outlet valve will eventually be closed”; “If the tank is full and a request is present, the outlet valve will eventually be opened”; “The program eventually terminates”.

Generally, given a transition system \mathcal{T} , we can ask the following questions:

- *Are undesired states reachable in \mathcal{T} , such as states that represent a deadlock, a violation of mutual exclusion, etc?*
- *Are there runs of \mathcal{T} such that, from some point on, some desired state can never be reached or some action can never be executed?* Such kind of runs represent livelocks where, for example, some process is prevented from entering its critical section, although other components of the system still make progress.
- *Is some initial state of \mathcal{T} reachable from every state?* In other words, can the system be reset?

However, we should keep in mind that the object under analysis is an abstract model. Possible counter-examples may be due to modeling artifacts and no longer correspond to the actual system runs. Reviews can thus be necessary to ensure that the abstract model reflects the behavior of the concrete system.

To sum up, model checking can be formally defined as follows:

Given a transition system \mathcal{T} and a formula φ , the model checking problem is to decide whether $\mathcal{T} \models \varphi$ holds or not. If not, the model checker should provide an explanation, in the form of a counter-example (i.e., a run of \mathcal{T} that violates φ). For this to be feasible, \mathcal{T} is usually required to be finite-state [Mer00].

Software Model Checking is a particular application of model checking, consisting of the algorithmic analysis of programs to prove properties of their execution [JM09].

In this approach a software system is seen as a state machine and so is modeled by a graph, consisting of:

- **nodes**, representing states of the system (e.g. value of program counter, variables, registers, stack/heap contents, and so on);

- **edges**, representing state transitions (e.g. events, input/output actions, internal steps).

The information (logical formulas) can be associated to either states or transitions. There are two kinds of models:

- Kripke structures (see Definition 48): information is placed in the states, called “atomic propositions”;
- Labeled Transition Systems (see Definition 49): information is placed in the transitions, called “action labels”.

The complexity of model checking in general and software model checking in particular, arises from:

- **State-space explosion**: the state-space of a software program is exponential on many parameters such as the number of variables.

This could become infinite in the presence of function calls and dynamic memory allocation. This *state-space explosion problem* must be addressed to allow real world problems to be solved. Model checking algorithms use the statements in the program to generate the set of states to be analyzed. These states need to be stored to ensure that they are visited at most once.

- **Expressive logics** (logics like Computation Tree Logic — CTL, and Linear Time Logic — LTL): have complex model checking algorithms.

There are different methods to deal with the problem of state-space exploration. Two of them will now be presented in some detail: symbolic model checking and bounded model checking. Other approaches, like *partial-order reduction* (ignoring some executions, because they are covered by others) and *on-the-fly execution* (integrating the model generation and verification phases, to prune the state space), are used by some of the tools reviewed in Section 3.5.

Symbolic model checking

Along the years, symbolic model checking [BCM⁺92, McM92], has proven to be a powerful technique to formally verify state systems such as sequential circuits and protocols. Since its definition in the early 90’s, it has been integrated in the quality assurance process of several hardware companies.

The ability to analyze systems of considerable size using model checking techniques requires efficient data structures to represent objects such as transition systems and sets of system states. A common representation in symbolic model checking algorithms is the Binary Decision Diagram (see Definition 47) [Bry86]. This kind of data structure is appropriate for the symbolic representation of sets because it offers the following features:

- Every boolean function has a unique and canonical BDD representation.
- Boolean operations (e.g. negation, conjunction, implication, etc) can be implemented with complexity proportional to the product of the inputs.
- The projection operation (quantification over one or several boolean variables) is easily implemented; in the worst case, its complexity is however exponential.

The use of BDDs within symbolic model checking techniques has however allowed for systems with 10^{20} states and more to be analyzed [BCM⁺92]. For the first time, a significant number of realistic systems could be verified, which led to a wider use of model checking techniques in industry. The following examples show that the use of symbolic model checking techniques has been successfully applied in different fields, enabling the discovery of bugs that were difficult to find with traditional techniques.

- In [CGH⁺95], the authors report how they applied this technique to verify the cache coherence protocol described in the *IEEE FutureBus+*. Various potential errors in the design of the protocol were found that had not been detected before.
- In [KLM91], the authors apply the same technique to check the cache consistency protocol of the *Encore Compute Corporation* distributed multiprocessor.
- In [DDHY92], it is reported how the cache coherence protocol of the *Scalable Coherent Interface* was verified and several errors found.
- In [CAB⁺98], a preliminary version of the system requirement specifications of the *Traffic Alert and Collision Avoidance System II* was verified.
- In [CGM⁺98], a railway interlocking system was also verified.

Despite the advantages referred previously, the bottleneck of symbolic model checking techniques is the amount of memory required for storing and manipulating BDDs. The boolean functions necessary to represent the set of states can grow exponentially. Numerous techniques such as decomposition, abstraction and reduction have been proposed throughout the years to tackle this problem but the full verification of many systems is still beyond the capacity of BDD-based symbolic model checkers.

Bounded model checking

Bounded Model Checking was introduced in 1999 by Biere et al [BCCZ99] as a complementary technique to the one present previously based on BDD. This method is called bounded because only states reachable within a bounded number k of steps are explored. Figure 3.14 gives a high-level overview of a bounded model checking technique. Essentially, the design under verification is unwound k times and joined together with a property to form a propositional formula, which is passed on to a Satisfiability Problem³ (SAT) solver. The formula is satisfiable if and only if there is a trace of length k that refutes the property. The technique is inconclusive if the formula is unsatisfiable, as there may be counter-examples longer than k steps.

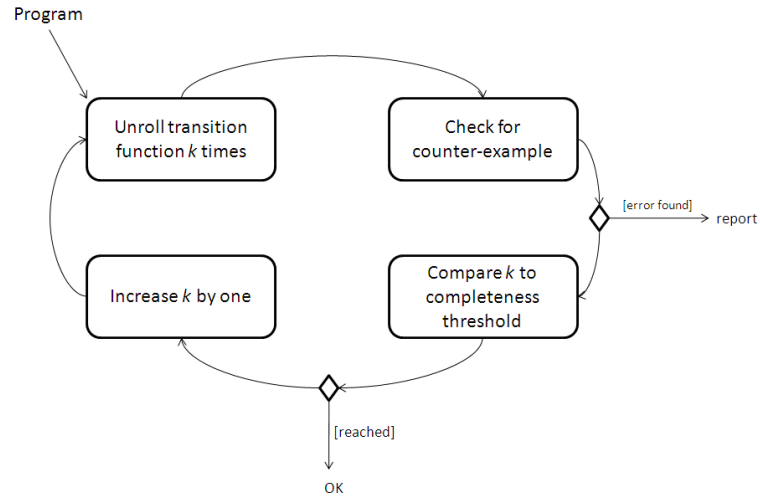


Figure 3.14: High level overview of Bounded Model Checking [DKW08]

Bounded model checking is defined in two main steps:

1. The sequential behavior of a transition system over a finite interval is encoded as a propositional formula.

This propositional formula is formed as follows: given a transition system M (which could be a Kripke structure), a temporal logic formula ϕ and a user-supplied time bound k , a propositional formula $\llbracket M, \phi \rrbracket_k$, which will be satisfiable if and only if the formula f is valid along some computation path of M .

2. The formula is given to a propositional decision procedure (a satisfiability solver), to obtain a satisfying assignment or to prove that there

³Recall that satisfiability is the problem of determining if the variables of a given formula can be assigned in such a way as to make the formula evaluate to true — Definition 23.

is none.

As claimed by the authors of the approach, the main advantages of this technique are the following:

- It finds counter-examples very quickly, due to the depth-first nature of SAT search procedures.
- It finds counter-examples of minimal length (contributing to a better understanding of the counter-example).
- It uses much less space than BDD-based approaches.
- Unlike from BDD-based approaches, bounded model checking does not need a manual selection of variable order.
- SAT procedures do not suffer from the space explosion problem of the BDD-based approaches (modern SAT solvers can handle propositional satisfiability problems with hundreds of thousands of variables or more). Experiments have shown that if k is small enough it outperforms BDD-based techniques.

The disadvantages of bounded model checking are:

- The user has to provide a bound on the number of cycles that should be explored. This implies that the method is incomplete if the bound is not high enough. Even though this can be seen as a disadvantage, BDD-based verification often requires a suitable, manual ordering for BDD variables, or certain to be carried out abstractions.
- The types of properties that can currently be checked are very limited. This means that one cannot be guaranteed a true or false decision for every specification.
- Despite the fact that the method may be extended, it has thus far only been used for specifications where fixpoint operations are easy to avoid.
- It has not been shown that the method can consistently find long counter-examples. This is because the length of the propositional formula subject to satisfiability solving grows with each step.

Despite the disadvantages of this method, bounded model checking has proved to be a valuable complement to verification techniques, being able to find bugs and sometimes to determine correctness, in situations where other verification techniques fail completely. During the last few years, there has been a major increase in the reasoning power of SAT solvers. This allowed bounded model checkers to handle instances with hundreds and thousands of variables. Symbolic model checkers, on the other hand, only can check systems with a few hundred of variables.

3.4 Dynamic Techniques

As referred previously, dynamic verification is a technique that is applied during the execution of software with the goal of checking its behavior at runtime (*dynamically*), aiming at finding bugs.

In this section, we will discuss two particular dynamic techniques: *run-time verification* and *testing*.

3.4.1 Runtime Verification

Runtime verification is an approach based on extracting information from a running system, using such information to detect behaviors satisfying or violating certain properties (possibly reacting to them). Currently, runtime verification techniques are often presented with various alternative names, such as runtime monitoring, runtime checking, runtime reflection, runtime analysis, dynamic analysis, runtime or dynamic symbolic analysis, trace analysis, log file analysis and so on. All these names referring to instances of the same high-level concept, but used in the context of different communities.

Runtime verification avoids the complexity of traditional formal techniques, such as model checking and theorem proving, by analyzing only a set of execution traces and by working directly with the actual system. Thus, this technique scales up relatively well when comparing it with model checking that suffers from the state-space explosion problem. However, in contrast to model checking, runtime verification has to deal with finite traces only, as at an arbitrary point in time, the system will be stopped and so its execution trace.

To check the desired properties, runtime verification specifications are usually expressed in predicate formalisms, such as finite state machines, regular expressions, context-free patterns, linear temporal logics, or extensions of these. However, in order to extract events from the program while it is running, it is necessary to insert these specifications in the source code. This technique is usually known as *code instrumentation*. Figure 3.15 depicts a view of a runtime monitor.

Aspect-Oriented Programming [KLM⁺97] (AOP) have been recently recognized as a technique for defining program instrumentation in a modular way. AOP generally promotes the modularization of crosscutting concerns.

Current challenges

One of the challenges posed to runtime verification is the runtime overhead. Extracting events from the executing system and sending them to monitors can generate a large runtime overhead if done naively. A good instrumenting system is critical for any runtime verification tool.

Also, when monitoring parametric properties, the monitoring system needs to keep track of the status of the monitored property with respect

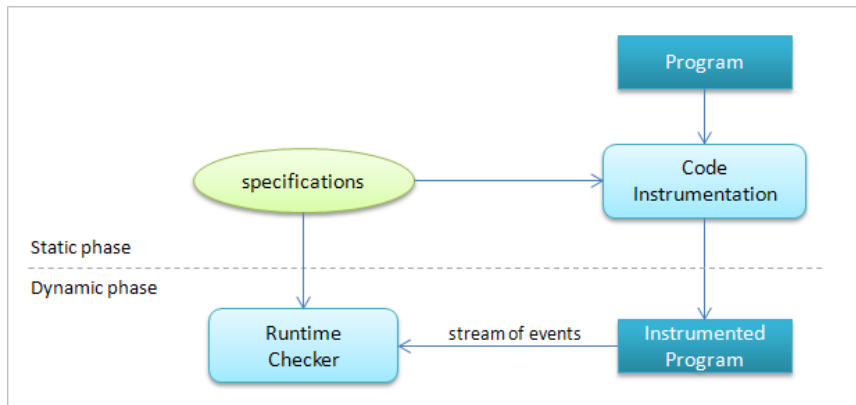


Figure 3.15: Runtime Monitor System.

to each parametric instance. The number of such instances is theoretically unbounded and tends to be enormous in practice. An important challenge is to find how to efficiently dispatch observed events to those instances which need them. Another challenge is how to keep the number of such instances small.

A recent approach to runtime verification is to use static analysis to reduce the amount of exhaustive monitoring. Static analysis can be performed both on the property to monitor and on the system to be monitored. With respect to the former, static analysis can reveal that: certain events are unnecessary to monitor; the creation of certain properties can be delayed; certain existing monitors will never trigger and thus can be garbage collected. With respect to the latter, static analysis can detect code that can never influence the monitors.

3.4.2 Testing

Testing, also known as *Experimentation*, is a phase of software verification that involves the execution of a system or a component. Basically, a number of test cases are chosen, where each test case consists of a set of test data.

The goal of this phase is to check that a software package:

- meets the business and technical requirements that guided its design and development;
- has the correct behavior (it works as expected).

Although the tests phase can be performed at any time in the development process, depending on the testing method employed (as will be discussed below), it usually takes place after the requirements have been defined and the coding process has been completed. *Test Driven Develop-*

ment differs from traditional models, as the effort of testing is on the hands of the developer and not in a formal team of testers (see Figure 3.16).

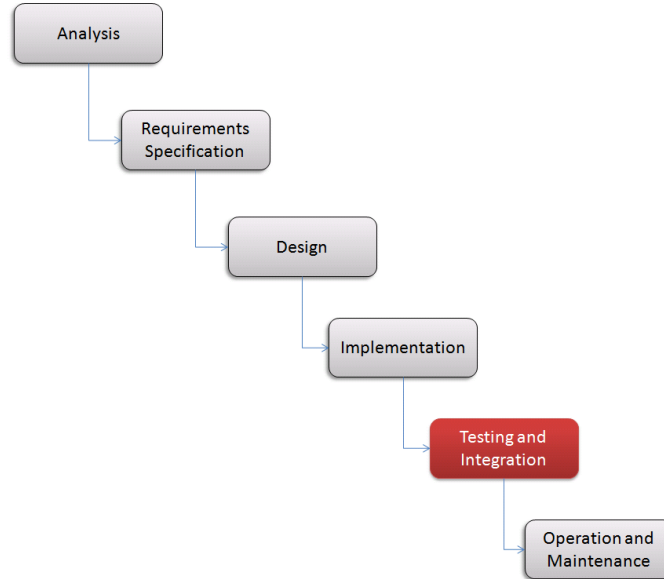


Figure 3.16: Test-driven Software development process.

3.4.3 Testing Methods

In the world of testing there are two predominant methodologies: *white-box* and *black-box*. These approaches describe the point-of-view taken by a test engineer when designing the test cases.

Black box

Black box testing (also called functional testing) *ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions* [Bei95, IEEE90].

This testing methodology looks at the available inputs for an application and the expected output for each one of these inputs. It is not concerned with the process used to achieve a particular output or any other internal aspect of the application that may be involved in the transformation of an input into an output. The basis of this strategy lies on the selection of appropriate data for each functionality and testing it against the functional specifications in order to check for normal/abnormal behavior of the application.

Basically, black box testing attempts to find errors in the external behavior of the code in the following categories:

- incorrect or missing functionality;

- interface errors;
- errors in data structures used by interfaces;
- behavior or performance errors;
- initialization and termination errors.

Through this aid of testing, one hopes to be able to determine if the program components work according to the specifications. However, it is important to remark that no amount of testing is in general capable of proving the absence of errors and defects in code.

One of the advantages of this method is its ease of use, as testers do not have to concern themselves with the inside of an application. The test only needs to be thorough with respect to the requirements specification, and to take into account how the system should behave in response to each particular action.

System Testing, *User Acceptance Testing*, and *Beta Testing* are examples of methods that fall in the category of black box testing.

White box

White box testing (also called structural testing or glass box testing) *takes into account the internal mechanisms of a system or component* [IEEE90].

According to Pressman [Pre86], with this testing technique, a software engineer can design tests that:

- Exercise independent paths within a module or unit;
- Exercise logical decisions on both their true or false side;
- Execute loops at their boundaries and within their operational bounds;
- Exercise internal data structures to ensure their validity.

With white box testing, the code should be run with predetermined inputs and it should be checked that it produces predetermined outputs. Frequently, programmers write stubs and drivers. A *driver* is a *software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results* [IEEE90] or more simplistically a line of code that calls a method and passes that method a value.

Essentially, the purpose of white box testing is to cover as many statements, decision points, and branches in the code base as possible.

Unit testing, *Integration testing* and *Regression testing* are examples of methods that falls in this category of white box testing.

The main difference between black box and white box testing is the area on which each of the methods focuses. While black box is focused on results (if an action is taken and it produces the desired result, the process to achieve that outcome is irrelevant), white box is focused on details (on the internal working of a system, that all paths have been tested and that the sum of all parts can contribute to the whole).

3.4.4 Testing Levels

Depending on the phase of the development process where they are performed and on their coverage, tests can be grouped in different levels corresponding to the grain of code to which they apply.

Unit testing

Unit testing is *testing of individual hardware or software units or groups of related units* [IEEE90].

Unit testing [Ham04] or *component testing* refers to tests that check the functionality of a specific piece of code. In object oriented programming languages, this unit corresponds to a class. The minimal unit test includes the constructors and destructors of the class.

The purpose of these unit tests is to check the conditional logic of one piece of code; usually they are written by developers that work on code (white box style method) to ensure that the functions behave as expected — they write test cases for each scenario of the module and the expected results.

Although a single unit test cannot check the functionality of a piece of software, it assures that the individual parts work well independently of each other. These tests help to find problems early in the development cycle.

Integration testing

Once the individual software modules have been tested, it is time to test them combined in groups — this phase is called *Integration Testing*.

Integration testing is *testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them* [IEEE90].

This phase takes place after Unit Testing and before System Testing. The idea is to take as input the modules that have been unit-tested before, group them in larger collections, apply them to tests defined in an integration plan test and deliver as output the integrated system, ready to the next phase — System Testing.

Integration testing has the goal of verifying the interfaces between components against a software design. These tests verify functional, performance

and reliability requirements of major design items. The tests are constructed to check that all components within a group interact correctly.

Classic integration testing strategies include:

- Top-down: an incremental technique that begins by testing the top level modules and progressively goes to the lower-level modules. Lower-level modules are normally simulated by stubs⁴ which mimic their functionality. As lower level code is added, stubs are replaced with the actual components. Top Down integration can be performed and tested in breadth first or depth first manner.

This technique has the advantage of provide early working module of the program and so design defects can be found and corrected early.

The disadvantage is that stubs need to be written very carefully as they will simulate setting of output parameters.

- Bottom-up: modules at the lowest levels are developed first and other modules which go towards the main program are integrated and tested one at a time. After the integration of lower-level modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. In this approach, lower-level modules are tested extensively, thus making sure that highest-level modules are tested properly.
- Big Bang: all or most of the modules are grouped to form a software system or a major part of the system, and are tested as a unit. This technique is very effective for saving time in the integration testing process when applied to small systems, but in larger systems it can prevent the testing team from achieving the goal of integration testing because it may be hard to tell in which subsystem the defect lies when a failure occurs. Once again, integration tests are white box tests performed by the programmers.

System Testing

System testing is one of the most important phases of the complete testing cycle.

System Testing is testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements [IEEE90].

System Testing evaluates the system compliance with specific functional and non-functional requirements (such as security, speed, accuracy, and re-

⁴A stub is a “false” program unit that stands for the real one, that is used (invoked) by another component under test, but is not yet coded; the stubs pretend to be the actual component.

liability) and should require no knowledge of the inner design of the code or logic (it is thus black box testing).

Essentially, this phase is an investigatory phase, where the focus is to test not only the design, but also the behavior, the bounds defined in the software/hardware requirements specification and according with expectations of the costumer.

There are different types of testing that should be considered during System testing, usually performed by specialized teams not including the programmers: GUI software testing, Usability testing, Performance testing, Compatibility testing, Error handling testing, Load testing, Volume testing, Stress testing (*testing conducted to evaluate a system or component at or beyond the limits of its specification or requirements* [IEEE90]), Security testing, Scalability testing, Sanity testing (tests that determine whether it is reasonable to proceed with further testing), Smoke testing (collection of tests that are performed on a system prior to being accepted for further testing), Exploratory testing, Ad hoc testing, Regression testing, Reliability testing, Installation testing, Maintenance testing, Recovery testing, and Accessibility testing.

Acceptance Testing

Once the application is ready to be released, the crucial step is *Acceptance Testing*, also called *User Acceptance Testing* (UAT).

Acceptance testing is formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a costumer) and to enable to determine whether or not to accept the system [IEEE90].

In this step, usually the final step before delivering the application, the end users are the testers. This type of testing gives them the confidence that the application being delivered meets their requirements. Real world scenarios and perceptions relevant to the end users are used in this step, which helps to find usability problems. A prerequisite to this phase is that the application should be fully developed. Previous levels of testing (Unit, Integration and System) are already completed before the User Acceptance Testing is done. Thus, at this level, most of the technical bugs have already been fixed.

To ensure a proper coverage of all the scenarios during testing, Test Cases are created. The steps taken for UAT typically involve one or more of the following:

- User Acceptance Test Planning: this planning outlines the UAT strategy, describes the key focus areas, entry and exit criteria.
- Designing User Acceptance Test Cases: this step helps to ensure that the UAT provides enough coverage of all the scenarios. Each Test Case

describes in a simple language the steps to be taken to test something.

- Selecting a Team that will execute the Test Cases: this team is usually a good representation of the real world end users.
- Executing Test Cases: the testing team executes the test cases and may also perform additional random tests.
- Documenting the bugs found during UAT: the testing team reports their comments and any defects or issues found during testing.
- Resolving the issues: the problems found during the testing are discussed and resolved with the consensus of end users.

Regression Testing

Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [IEEE90].

In other words, each time that a major code change occurs, regression testing [KFN99] should be performed. It is a kind of white box integration test.

A way to do this is by rerunning existing tests against the modified code to determine whether the changes break anything that worked prior to the change and by writing new tests where necessary. According to Beizer [Bei90], the repetition of tests should reveal that the software's behavior is unchanged.

One of the main reasons for regression testing is that it is often extremely difficult for a programmer to understand how a change in one part of the software will be echoed in other parts of the software. So, there are some factors to consider during the process of regression testing. These include the following (but are not limited to):

- Testing fixed bugs promptly.
- Watching for side effects of fixes — frequently, a fix for a problem in one place inadvertently causes a software bug in another place.
- Writing a regression test for each bug fixed — in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program [HK].
- Making changes (small and large) to data and finding any resulting corruption.

Beta Testing

Beta testing is operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs, and fits within the business process.

Beta testing is often employed as a form of external acceptance testing for off-the-shelf software in order to acquire feedback from the market. The advantages of running beta tests are [Gal04]:

- *Identification of unexpected errors* because the beta testers use the software in different and unexpected ways.
- *A wider population search for errors* in a variety of environments (e.g. different operating systems).
- *Low costs* because the beta testers generally obtain free software but are not otherwise compensated.

The disadvantages of running beta tests are [Gal04]:

- *Lack of systematic testing* because each user uses the product in any manner, at their choice.
- *Low quality error reports* because the users may not report errors or these reports do not have enough detail.
- *Much effort is necessary to examine error reports* when there are many beta testers.

Beta testing is important because it is almost impossible for developers to test their software in all of the various conditions that might occur.

In term of its ability to guarantee software correctness, runtime verification is weaker than formal methods but stronger than testing. Testing can only guarantee the correctness of a limited set of inputs at implementation time and it does not guarantee that the system will operate as expected under untested inputs. Runtime verification allows for formal specification and verification/testings of the properties that a system has imperatively to satisfy.

3.5 Tools

After presenting an overview of software verification and the theoretical foundations behind it, as well as the current approaches to this relevant but complex topic, it is time to survey tools serving that purpose following the different methods referred, in order to make clear that there is still scope for new tools, in particular combining techniques that have not been combined before.

3.5.1 Program Verifiers

The list of tools presented in this section does not pretend to be an exhaustive list of the existent program verifiers. Only the most relevant ones for the work presented in this document are referred.

ESC/Java2

ESC/Java2 (formerly called ESC/Java) [FLL⁺02], that stands for Extended Static Checking for Java, is a program checker, pioneered by *Flanagan et al*, that attempts to find runtime errors in Java programs through the static analysis of the source code. It is *static* because the checking is performed without running the program and *extended* because it catches more errors that are caught by conventional static checkers such as type checkers.

This checker is powered by a VCGen and takes advantage of automatic theorem proving techniques to reason about the semantics of programs. This allows ESC/Java2 to give warnings about many errors that are typical in modern programming languages (null dereferences, array bound errors, type cast errors, and so on) and also warns about synchronization errors in concurrent programs. The architecture of ESC/Java2 is depicted in Figure 3.17.

The annotation language used by ESC/Java2 is JML-based, but there are some differences between the two languages, at both the syntactic and the semantic levels. While JML is intended to allow full specification of programs, ESC/Java2 is intended only for light-weight specification. But they have in common the fact the annotations appear like other Java declarations, modifiers, or statements, but enclosed in Java comments that begin with an @-sign.

ESC/Java2's front-end acts like a normal Java compiler, but parses and type checks ESC/Java2 annotations as well as Java source code. This front-end produces an *abstract syntax tree* and a formula in first-order logic encoding information about the types and fields that procedures/methods in that class use. After parsing and producing these intermediate results, the front-end translates each procedure body into a simple language based on Dijkstra's Guarded Commands.

The next step is to generate the verification conditions for the resulting guarded commands. The computation of these VCs is similar to the computation of a weakest precondition, but ESC/Java2 VC-generation includes optimizations to avoid the potential exponential growth inherent to a naive weakest precondition computation [FS01, LMS08].

For each procedure, the next step invokes an automatic theorem prover (Simplify) to check which VCs are provable or not. In the last step, the post-processor processes the theorem prover's output, producing warnings when the prover is unable to discharge the verification conditions. When Simplify fails to prove some VC, it finds and reports one or more counterexamples.

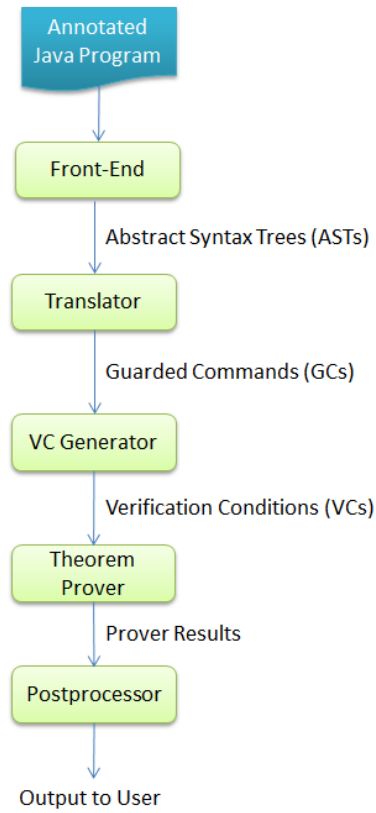


Figure 3.17: ESC/Java2 architecture.

Although the analysis done by ESC/Java2 is neither *sound* nor *complete*, this was intentional (in a pragmatic sense) as a way to reduce the number of errors and/or warnings reported to the programmer and make ESC/Java2 a useful tool in practice. However, as a result of this imprecise analysis the programmer may wrongly deduce that a correct program is incorrect (*false positives*) or a wrong program can be erroneously considered correct (*false negatives*).

Frama-C

Frama-C⁵, which stands for *Framework for Modular Analysis of C programs*, is a set of *static* analyzers for C programs. The architecture of Frama-C, which is depicted in Figure 3.18, is similar to the Eclipse architecture, in the sense that it relies on a set of plugins. Due to the fact that the AST is the core of Frama-C, it can be freely manipulated by different kinds of other plugins. They can be divided into two major groups:

⁵ Available at <http://frama-c.com>

1. *Code Analysis* — it is possible to run the following analyzes over the source code:
 - *Impact Analysis* — highlights the locations in the source code that are impacted by a given modification.
 - *Scope and Dataflow Analysis* — allows the user to navigate the dataflow of the program, from definition to use or from use to definition.
 - *Variable Occurrence Browsing* — allows the user to reach statements where a given variable is used.
 - *Value analysis* [CCM09] — this plugin allows the computation of variation domains for variables. It uses abstract interpretation techniques.
 - *Jessie* — a deductive verification plugin based on weakest precondition techniques. It allows to prove that the source code is in accordance with its specification (written in ANSI/ISO C Specification Language — ACSL [BFH⁺08]), as explained below.
2. *Code Transformation* — it is possible to do the following transformations over the source code:
 - *Semantic Constant folding* — using the results of the value analysis, this plugin replaces in the source code constant expressions by their values. As it is based on semantic analysis, it is able to do more of these simplifications than a syntactic analysis would do.
 - *Slicing* — given a slicing criterion, this plugin creates a copy of the program and slices off those parts which are not necessary according to that criterion.
 - *Spare code* — removes code that does not contribute to the final results of the program.

With respect to the Jessie VCGen, in order to check whether a program is correct or not, in a first step, the **Frama-C** core processes the source code in order to produce an abstract syntax tree in the **C Intermediate Language (CIL)** format. The source code can possibly include annotations written in **ACSL**, and this is reflected in the **AST** (if these annotations are not provided, *Jessie* plugin can infer them [MM11]). Given the **AST** and its annotations, the *Jessie* plugin produces code in its own format. From *Jessie* code, *Why* [FM07] code is produced using the *Jessie2Why* tool. The *Why* VCGen reads this code and produces a set of proof obligations which are exported to either automatic provers or proof assistants.

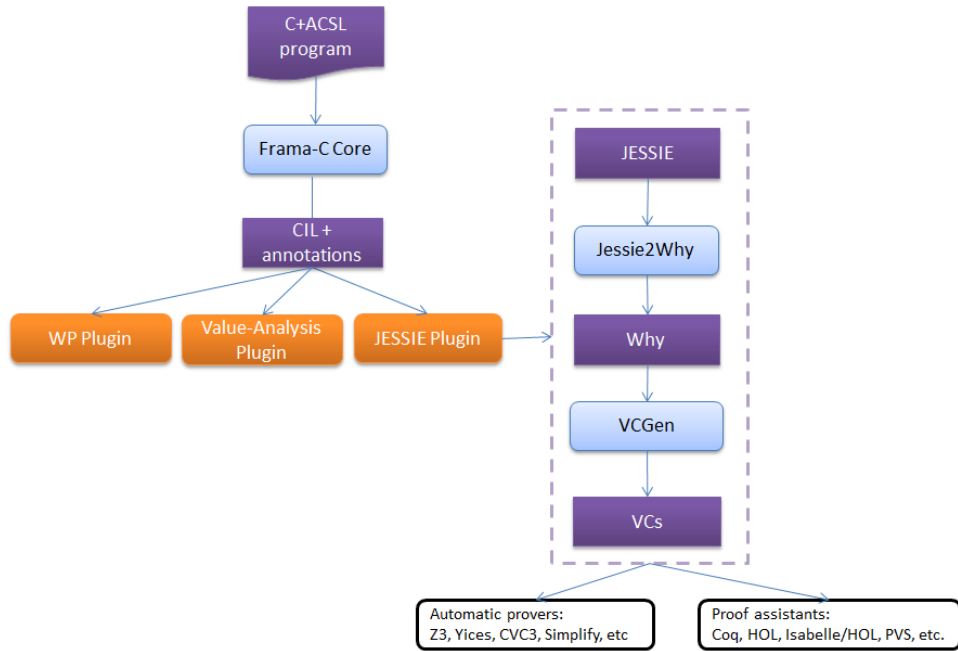


Figure 3.18: Frama-C architecture

Boogie

Boogie [BCD⁺05] is a program verification system that produces verification conditions for programs written in an intermediate verification language, also called Boogie. These verification conditions are then passed to the Z3 theorem prover, outputting as result an indication of whether the program is correct or incorrect (in this case, it points out which annotations were violated) or a timeout.

The architecture of Boogie is depicted in Figure 3.19. As can be seen, Boogie has multiple front-ends for C#, C enriched with annotations, Dafny [Lei10] (similarly to Boogie, Dafny is both an imperative object-based language and a verifier) and Chalice [LMS09] (a language for specification and verification of concurrency properties of programs). The Boogie language is a simple language with procedures whose implementations are basic blocks consisting mostly of four kinds of statements: assignments, asserts, assumes, and procedure calls [LSS99].

Spec#

Spec# [BDF⁺08] is an extension of the object oriented language C# providing contracts to be added to methods, in the form of pre and postconditions as well as invariants. Spec# is also a compiler that statically enforces non-null types and emits runtime checks for contracts and invariants. The

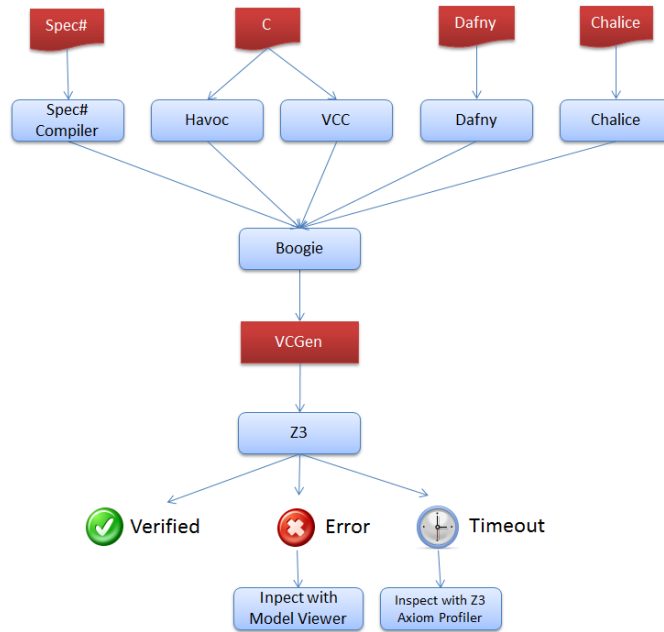


Figure 3.19: Boogie architecture

architecture of **Spec#** is implicitly shown in Figure 3.19. The **Spec#** programming system consists of the language, the compiler and the Boogie verifier.

Spec# is different from standard compilers due to the fact that it does not only produce executable code from a program written in the **Spec#** language, but it also translates all specifications into a language-independent format. Having the specifications available as separate compiled units allows program analysis and verification tools to consume the specifications without the need to either modify the **Spec#** compiler or to write a new source-language compiler. The **Spec#** compiler targets the Microsoft .NET Common Language Runtime — CLR [BP02] and attaches a serialized specification to each program component for which a specification exists. It is due to this fact that Boogie consumes compiled code rather than source code.

Code Contracts [FBL10] in the .NET Framework 4.0 have evolved with **Spec#**. Code Contracts provides a language-agnostic way to express coding assumptions in .NET programs, and is still a work in progress.

VCC

VCC [CDH⁺09], which stands for Verifying C Compiler, is a tool that aims at proving the correctness of *annotated concurrent C programs*. The architecture of VCC is depicted in Figure 3.20.

Essentially, VCC accepts annotated C programs, and translates them

into a set of logical formulas using the **Boogie** tool, which in turn are sent to an automated SMT solver — **Z3** [dMB08b] — to check their validity. If **VCC** reports that it is unable to verify the correctness of one or more of the formulas, the **Model Viewer** can be used to inspect how **VCC** thinks the program might fail. If **VCC** reports a timeout then the **Z3 Axiom Profiler** can be used to see which part of the program is causing the verifier to timeout and where the prover is spending its time.

VCC was built thinking in the verification of **Hyper-V**, a layer of software that sits directly on hardware and consists of 100 KLOC of kernel-level C and x64 assembly code.

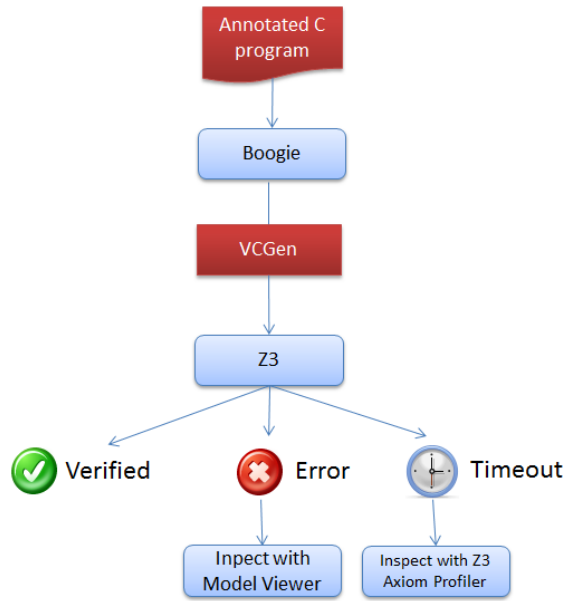


Figure 3.20: VCC architecture

3.5.2 Static Analysis Tools

Goanna

Goanna [RHJ⁺06] is a static analysis tool for C/C++ source code based on model checking. It uses the **NuSMV** [CCGR00] model checker as the underlying verification engine, allowing the specification of user defined properties.

The CTL (Computation Tree Logic) model checking problem under the **Goanna** tool is encoded in two. First, the atomic propositions of interest for reasoning are defined, e.g., whether a variable is declared, used, or assigned a value. For instance, for a variable named x these properties are respectively $declx$, $usedx$ and $assignedx$. A pattern matching approach is used to relate certain patterns in a program's abstract syntax tree with propositions

of interest. In a second step the control flow graph of a program is automatically extracted and labeled with the previously determined propositions.

GCC is used as a front end in the **Goanna** tool, as one of its features is that it allows the AST of C/C++ programs to be output in an intermediate language. The tool parses the AST and then on the one hand generates the CFG from it, and on the other hand it matches patterns in the AST, which constitute the atomic propositions of a CTL formula expressing the desired property. The CFG is labeled with atomic propositions where their respective patterns were matched. Once the patterns and the CTL formula have been specified, the translation of the C/C++ source code into a suitable NuSMV model and its checking are fully automatic.

Splint

Splint [EL02] is a lightweight static analysis tool for ANSI C. **Splint** statically checks code by exploiting annotations⁶ added to libraries and programs that document assumptions and intents. **Splint** finds potential vulnerabilities by checking whether the source code is consistent with the properties implied by the annotations.

As the main goal of **Splint** is to do as much useful checking as possible, it is both unsound and incomplete. **Splint** thus produces both false positives and false negatives. The intention is that the warnings are useful to programmers, but no guarantee is offered that all messages indicate real bugs or that all bugs will be found. The tool can also be configured to suppress particular messages and to weaken or strengthen assumption checking.

In order to make analysis fast and scalable to large programs, **Splint** assumes certain compromises. The most important is to limit the analysis to data flow within procedure bodies. **Splint** analyzes procedure calls using information from annotations that describe preconditions and postconditions. Another compromise is that between flow-sensitive analysis, which considers all program paths, and flow-insensitive analysis, which ignores control flow (see section 2.2 for a discussion of strategies).

Splint analyzes loops using heuristics to recognize common idioms. This allows it to correctly determine the number of iterations and bounds of many loops without requiring loop invariants or abstract evaluation. **Splint** also detects both stack and heap-based buffer overflow vulnerabilities [LE01]. The simplest detection techniques just identify calls to often misused functions; more precise techniques depend on function descriptions and program-value analysis.

In addition to the built-in checks, **Splint** provides mechanisms for defining new checks and annotations to detect new vulnerabilities or violations of application-specific properties.

⁶The annotations refers to stylized C comments identified by an @ character following the /* comment marker.

Coverity Prevent

Coverity Prevent⁷ identifies security vulnerabilities and code defects in C, C++, C#, and Java code. Coverity Prevent discovers code defects using a combination of inter-procedural data flow analysis and statistical analysis techniques [ALS06].

In the case of inter-procedural data flow analysis, Coverity Prevent analyzes each function and generates a context-sensitive summary for it. Each summary describes all characteristics of the function that have important implications for externally-observable behavior. Coverity Prevent calls these summaries *function models*. During data flow analysis, these summaries are used to determine the effects of function calls whenever possible. The tool also performs false path pruning to eliminate infeasible paths from consideration. This reduces computation costs and lowers the chances of generating false positives.

In the case of statistical analysis, Coverity Prevent uses statistical inference techniques to detect important trends in the code. Using these conclusions, it then tries to discover statistically significant coding anomalies, which may represent deviations from the software developers' intentions.

3.5.3 Model Checking Tools

SPIN

SPIN [BA08] is a model checker whose main goal is to verify the correctness of distributed software models. Initially it was used for the verification of temporal logic properties of communication protocols specified in the Promela language. Promela supports simple data-types, non-deterministic assignment and conditionals, simple loops, thread creation, and message passing.

SPIN generates dedicated C source code for checking each model in a way that saves memory and improves performance. Furthermore, it offers the following options to speed up the model-checking process:

- Partial order reduction;
- State compression;
- Bitstate hashing (a technique that instead of storing whole states, only stores their hash codes in a bit field).

Java Pathfinder

Java Pathfinder (JPF) [BHPV00] is an execution-based model checker for Java programs that modifies the Java Virtual Machine to implement a systematic search over different thread schedules.

⁷<http://www.coverity.com>

In a first version, JPF translated **Java** code to **PROMELA** and used **SPIN** for model checking [HP98]. However, the main problem of **Promela** relies on the fact that it does not support dynamic memory allocation and therefore is not well suited to modeling the **Java** language. So, to fill that gap, recent versions of JPF skip the translation phase and analyze the **Java** bytecode directly, and also handles a much larger class of **Java** programs than the first implementation. This new approach brings significant advantages:

- The use of the **JVM** makes it possible to store the visited states, which allows the model checker to use many of the standard reduction-based approaches (partial order, abstraction, etc) to prevent state-explosion.
- As the visited states are stored, the model checker can use different search-order heuristics without being limited by the requirements of stateless search.
- The possibility to use techniques such as symbolic execution and abstraction to compute inputs that force the system into states that are different from those previously visited allows for a high level of coverage.

JPF has been successfully used inside **NASA** to find subtle errors in several **Java** components [BDG⁺04b].

SLAM

SLAM [BCLR04, BBC⁺06] automatically checks that a **C** program correctly uses the interface to an external library. The **SLAM** analysis engine forms the core of a new tool called **Static Driver Verifier (SDV)** that systematically analyzes the source code of Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows operating system kernel.

The basic idea is that checking a simple rule against a complex **C** program (such as a device driver) should be possible by simplifying the program to make the analysis tractable. That is, it should be possible to find an abstraction of the original **C** program that has all of the behavior of the original program (plus additional behavior that is not relevant for checking the rule of interest). Boolean programs [BR00b, BR00c] are the **SLAM** solution for such an abstraction.

The **SLAM** automated process can be split in three steps: abstraction, checking and refinement. In the first step, given a **C** program \mathcal{P} and a set of predicates \mathcal{E} , the goal is to efficiently construct a precise Boolean program abstraction \mathcal{B} of \mathcal{P} with respect to \mathcal{E} . In the second step, given a Boolean program \mathcal{B} and an error state, the goal is to check whether or not the error state is reachable. Finally, in the third step, if the Boolean program \mathcal{B} contains an error path and this path is a feasible execution path

in the original program \mathcal{P} , then the process has found a potential error. If this path is not feasible in \mathcal{P} then the Boolean program \mathcal{B} is refined so as to eliminate this false error path.

SLAM comprises the predicate abstraction tool C2BP [BPR01, BMMR01] and the BDD-based model checker BEBOP [BR00a] for Boolean programs.

BLAST

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) [HJMS03, BHJM07] is an automatic verification tool for checking temporal safety properties of C programs. The task addressed by BLAST is checking whether software satisfies the behavioral requirements of its associated interfaces. BLAST employs Counter-Example-Guided-Automatic-Abstraction (CEGAR) refinement to construct an abstract model that is then model-checked for safety properties. The abstraction is constructed on the fly, and only up to the requested precision.

Given a C program and a temporal safety property, BLAST either statically proves that the program satisfies the safety property, or provides an execution path that exhibits a violation of the property (or, since the problem is undecidable, does not terminate) [HJMS03].

Like SLAM, BLAST provides a language to specify reachability properties. Internally, C programs are represented as Control Flow Automata (CFA), which resemble control flow graphs except that operators are placed on the edges rather than vertices. The BLAST lazy abstraction algorithm is composed of two phases. In the forward-search phase a reachability tree is built, which represents a portion of the reachable, abstract state space of the program. Each node of the tree is labeled by a vertex of the CFA and a formula, called the reachable region, constructed as a boolean combination of a finite set of abstraction predicates.

Initially the set of abstraction predicates is empty. The edges of the tree correspond to edges of the CFA and are labeled by basic program blocks or assume predicates. The reachable region of a node describes the reachable states of the program in terms of the abstraction predicates, assuming execution follows the sequence of instructions labeling the edges from the root of the tree to the node.

If the algorithm finds that an error node is reachable in the tree, then it goes to the second phase, which checks if the error is real or a result of the abstraction being too coarse. In the latter case, a theorem prover is asked to suggest new abstraction predicates which rule out that particular spurious counterexample. The program is then refined locally by adding the new abstraction predicates only in the smallest subtree containing the spurious

error; the search continues from the point that is refined, without touching the part of the reachability tree outside that subtree.

VeriSoft

VeriSoft [God97] is a verification tool that tries to avoid state explosion by discarding the states it visits. VeriSoft does not store the visited states, allowing to repeatedly visit them and explore them. This method is *stateless* and has to limit the depth of its search to eschew non-termination.

This tool was built for systematically exploring the state space of systems composed of several concurrent processes executing arbitrary C/C++ code. The purpose of the tool is to automatically detect coordination problems between concurrent processes.

Essentially, VeriSoft takes as input the composition of several Unix processes that communicate by means of message queues, semaphores and shared variables that are visible to the VeriSoft scheduler. The scheduler traps calls made to access the shared resources, and by choosing which process will execute at each trap point, the scheduler is able to exhaustively explore all possible interleavings of the execution.

The approach used by VeriSoft is however incomplete for transition systems that contain cycles.

SATABS

SATABS [CKSY04] is a model checking verification tool that uses a SAT-solver to construct abstractions and for symbolic simulation of counterexamples. This tool automatically generates and checks proof conditions for array bound violations, invalid pointer dereferencing, division by zero, and assertions provided by the user.

SATABS uses the SAT-based model checker Boppo [CKS05] to compute the reachable states of the abstract program. Boppo relies on a Quantified Boolean Formula (QBF) solver for fixed-point detection.

SATABS can verify concurrent programs that communicate via shared memory.

CBMC

CBMC [CKY03] is a tool that implements the Bounded Model Checking technique. It emulates a wide range of architectures and environments for the design under test. It supports both little and big Endian memory models, as well as header files needed for Linux, windows and Mac-OSX.

The main application of CBMC is for checking consistency of system-level circuit models given in C or SystemC with an implementation given in Verilog.

Saturn

Saturn [XA05] is a specialized implementation of BMC tailored to the properties it checks. The authors of the tool have applied it to check two properties of Linux Kernel code: Null-pointer dereferences and locking API conventions. With these tests they have shown that the technique is sufficiently scalable to analyze the entire Linux kernel. Soundness is relinquished for performance: **Saturn** performs at most two unwindings of each loop (so bugs that require more than two unwindings are missed).

Cmc

Cmc [MPC⁺02] is an execution based model checker for C programs that explores different executions by controlling schedules at the level of the OS scheduler. **Cmc** stores a hash of each visited state. **Cmc** has been used to find errors in implementations of network protocols [ME04] and file systems [YTEM06].

Chapter 4

State-of-the-Art: Slicing

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

René Descartes, 1596 — 1650

Since Weiser first proposed the notion of slicing in 1979 in his PhD thesis [Wei79], hundreds of papers have been proposed in this area. Tens of variants have been studied, as well as algorithms to compute them. Different notions of slicing have different properties and different applications. These notions vary from Weiser’s syntax-preserving static slicing to amorphous slicing which is not syntax-preserving; algorithms can be based on dataflow equations, information flow relations or dependency graphs.

Slicing was first developed to facilitate program debugging [Mar93, ADS93, WL86], but it is then found helpful in many aspects of the software development life cycle, including software testing [Bin98, HD95], software metrics [OT93, Lak93], software maintenance [CLM96, GL91b], program comprehension [LFM96, HHF⁺01], component re-use [BE93, CLM95], program integration [BHR95, HPR89b] and so on.

In this chapter, slicing techniques are presented including static slicing, dynamic slicing and the latest slicing techniques. We also discuss the contribution of each work and compare the major difference between them.

Structure of the chapter. In Section 4.1 is presented the concept of program slicing and its variants. The relationship among the different slicing techniques and the basic methods used to pose program slicing in practice are also discussed. In Section 4.2 the basic slicing approaches are presented. In Section 4.3 is reviewed the non-static slicing approaches. In Section 4.4 is reviewed the applications of program slicing. In Section 4.5 are presented some tools using the program slicing approach.

4.1 The Concept of Program Slicing

In this section it is presented the original static slice definition and also its most popular variants. At the end of each subsection, the respective concept will be clarified through. The examples are based on the program introduced hereafter in subsection 4.1.1.

4.1.1 Program Example

Listing 4.1 below corresponds to a program, taken from [CCL98], that will be used as the running example for all the next subsection aiming at illustrating each concept introduced. That program takes the integers n , $test$ and a sequence of n integers a as input and compute the integers $possum$, $posprod$, $negsum$ and $negprod$. The integers $possum$ and $negsum$ accumulate the sum of the positive numbers and of the absolute value of the negative numbers in the sequence, respectively. The integers $posprod$ and $negprod$ accumulate the products of the positive numbers and the absolute value of the negative numbers in the sequence, respectively. Whenever an input a is zero, the greatest sum and the greatest product are reset if the value of $test$ is non zero. The program prints the greatest sum and the greatest product computed.

```

2  main() {
   int a, test, n, i, posprod, negprod, possum, negsum, sum, prod;
   scanf("%d",&test); scanf("%d",&n);
4   i = posprod = negprod = 1;
   possum = negsum = 0;
6   while (i <= n) {
       scanf("%d",&a);
8       if (a > 0) {
           possum += a;
10          posprod *= a;
       }
12      else if (a < 0) {
           negsum -= a;
14          negprod *= (-a);
       }
16      else if (test) {
           if (possum >= negsum) {
18              possum = 0;
           }
20          else { negsum = 0; }
           if (posprod >= negprod) {
22              posprod = 1;
           }
24          else {
               negprod = 1;
           }
26      }
28      i++;
30      if (possum >= negsum) {
           sum = possum;
32      }
       else { sum = negsum; }
34      if ( posprod >= negprod) {
           prod = posprod;
36      }
       else { prod = negprod; }
38      printf("Sum: %d\n", sum);
       printf("Product: %d\n", prod);
40 }

```

Listing 4.1: Program example

4.1.2 Static Slicing

Program slicing, in its original version, is a decomposition technique that extracts from a program the statements relevant to a particular computation. A **program slice** consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*.

Definition 53. A static slicing criterion of a program P consists of a pair $C = (p, V_s)$, where p is a statement in P and V_s is a subset of the variables in P .

A slicing criterion $C = (p, V_s)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V_s .

Definition 54. Let $C = (p, V_s)$ be a static slicing criterion of a program P and $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \dots, (p_k, \sigma_k) \rangle$ a state trajectory of P on input I . $\forall i, 1 \leq i \leq k$:

$$Proj'_C(p_i, \sigma_i) = \begin{cases} \lambda & \text{if } p_i \neq p \\ \langle (p_i, \sigma_i|_{V_s}) \rangle & \text{if } p_i = p \end{cases}$$

where $\sigma_i|_{V_s}$ is σ_i restricted to the domain V_s , and λ is the empty string.

The extension of $Proj'$ to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$Proj_C(T) = Proj'_C(p_1, \sigma_1) \dots Proj'_C(p_k, \sigma_k)$$

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projections in its behavior.

Definition 55. A static slice of a program P on a static slicing criterion $C = (p, V_s)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I , with state trajectory T , then P' also halts, with the same input I , with the trajectory T' , and $Proj_C(T) = Proj_C(T')$.

The task of computing program slices is called *program slicing*.

Weiser defined a program slice S as a reduced, *executable program* obtained from a program P removing statements, such that S preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point.

Executable means that the slice is not only a closure of statements, but also can be compiled and run. *Non-executable* slices are often smaller and thus more helpful in program comprehension.

The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward traversal* of the program, starting at the slicing criterion. Therefore, these slices are referred to as **backward slices** [Tip95]. In [BC85], Bergeretti and Carré were the first to define a notion of a *forward slice*. A **forward slice** is a kind of ripple effect analysis, this is, it consists of all statements and control predicates dependent on the slicing criterion. A statement is dependent of the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine if the statement under consideration is executed or not.

Both *backward* or *forward* slices are classified as *static* slices. **Static** means that only statically available information is used for computing slices, this is, all possible executions of the program are taken into account; no specific input I is taken into account.

Since the original version proposed by Weiser [Wei81], various slightly different notions of program slices, which are not static, have been proposed, as well as a number of methods to compute slices. The main reason for this diversity is the fact that different applications require different program properties of slices.

The last two concepts presented (*dicing* and *chopping* in section 4.1.9 and 4.1.10, respectively) are two variations on the slicing theme but very related to slicing.

Listing 4.2 emphasizes the variable *sum* (in red color) and the variables affect by its value (in blue color).

At a first glance, if we only focus at variable *sum* in program of Listing 4.1 it is easy to infer that its value depends on values of *possum* and *negsum*. Listing 4.3 shows a static slice of the program in Listing 4.1 on the slicing criterion $C = (38, \textit{sum})$ ¹.

4.1.3 Dynamic Slicing

Korel and Laski [KL88, KL90] proposed an alternative slicing definition, named *dynamic slicing*, where a slice is constructed with respect to only one execution of the program corresponding just to one given input. It does not include the statements that have no relevance for that particular input.

Definition 56. A dynamic slicing criterion of a program P executed on input I is a triple $C = (I, p, V_s)$ where p is a statement in P and V_s is a subset of the variables in P .

Definition 57. A dynamic slice of a program P on a dynamic slicing criterion $C = (I, p, V_s)$ is any syntactically correct and executable program P'

¹Whenever not ambiguous, statements will be referred by their line numbers.

obtained from P by deleting zero or more statements, and whenever P halts, on input I , with state trajectory T , then P' also halts, on the same input I , with state trajectory T' , and $\text{Proj}_{(p, V_s)}(T) = \text{Proj}_{(p, V_s)}(T')$.

```

main() {
2   int a, test, n, i, posprod, negprod, possum, negsum, sum, prod;
   scanf("%d",&test); scanf("%d",&n);
4   i = posprod = negprod = 1;
   possum = negsum = 0;
6   while (i <= n) {
       scanf("%d",&a);
       if (a > 0) {
8           possum += a;
           posprod *= a;
10          }
       else if (a < 0) {
12           negsum -= a;
           negprod *= (-a);
14          }
       else if (test) {
           if (possum >= negsum) {
16               possum = 0;
18           }
           else { negsum = 0; }
           if (posprod >= negprod) {
20               posprod = 1;
22           }
           else {
24               negprod = 1;
26           }
       }
       i++;
28   }
   if (possum >= negsum) {
30       sum = possum;
32   }
   else { sum = negsum; }
34   if ( posprod >= negprod) {
       prod = posprod;
36   }
   else { prod = negprod; }
38   printf("Sum: %d\n", sum);
   printf("Product: %d\n", product);
40 }

```

Listing 4.2: Program with *sum* variable emphasized

```

main() {
2   int a, test, n, i, possum, negsum, sum;
   scanf("%d",&test); scanf("%d",&n);
4   i = 1;
   possum = negsum = 0;
6   while (i <= n) {
       scanf("%d",&a);
       if (a > 0) {
8           possum += a;
10          }
       else if (a < 0) {
12           negsum -= a;
14          }
       else if (test) {
           if (possum >= negsum) {
16               possum = 0;
18           }
           else { negsum = 0; }
20          }
       i++;
22   }
   if (possum >= negsum) {
       sum = possum;
24   }
   else { sum = negsum; }
26   printf("Sum: %d\n", sum);
}

```

Listing 4.3: A static slice of program in Listing 4.1

Due to run-time handling of arrays and pointer variables, dynamic slicing treats each element of an array individually, whereas static slicing considers each definition or use of any array element as a definition or use of the entire array [JZR91]. Similarly, dynamic slicing distinguishes which objects are pointed to by pointer variables during a program execution.

In section 4.3, this concept of *dynamic slicing* will be detailed and algorithms for its implementation will be also presented.

Listing 4.4 shows a dynamic slice of the program in Listing 4.1 on the slicing criterion $C = (I, 38, \text{sum})$ where $I = \langle (test, 0), (n, 2), (a_1, 0), (a_2, 2) \rangle$ ².

```

1 main() {
2     int a, test, n, i, possum, negsum, sum;
3     scanf("%d", &test); scanf("%d", &n);
4     i = 1;
5     possum = negsum = 0;
6     while (i <= n) {
7         scanf("%d", &a);
8         if (a > 0) {
9             possum += a;
10        }
11        i++;
12    }
13    if (possum >= negsum) {
14        sum = possum;
15    }
16    printf("Sum: %d\n", sum);
17 }
```

Listing 4.4: A dynamic slice of program in Listing 4.1

In the first loop iteration, hence the value of a is zero and so none of the statements in the if expression is executed, the whole conditional branch is excluded from the program slice. However, at the second loop iteration, the if statement is executed and so is included in the program slice, being excluded the else statements and the last else if statements.

Notice that the dynamic slicing in Listing 4.4 is a subprogram of the static slice in Listing 4.3 — all the statements that will never be executed, under the values of the input, are excluded.

4.1.4 Quasi-static Slicing

Venkatesh introduced in 1991 the *quasi-static* slicing in [Ven91], which is a slicing method between static slicing and dynamic slicing. A quasi-static slice is constructed with respect to some values of the input data provided to the program. It is used to analyze the behavior of the program when some input variables are fixed while others vary.

Definition 58. A quasi-static slicing criterion of a program P is a quadruple $C = (V'_i, I', p, V_s)$ where p is a statement in P ; V_i is the set of input variables of a program P and $V'_i \subseteq V_i$; and I' is the input data just for the subset of variables in V'_i .

²The subscripts refer to different occurrences of the input variable a within the different loop iterations.

Definition 59. A quasi-static slice of a program P on a quasi-static slicing criterion $C = (V'_i, I', p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I , with state trajectory T , then P' also halts, on input I , with state trajectory T' , and $Proj_{(p,V)}(T) = Proj_{(p,V)}(T')$.

Definition 60. Let V_i be the set of input variables of a program P and $V'_i \subseteq V_i$. Let I' be an input for the variables in V'_i . A completion I of I' is any input for the variables in V_i , such that $I' \subseteq I$.

Each completion I of I' identifies a trajectory T . We can associate I' with the set of trajectories that are produced by its completions. A quasi-static slice is any subset of the program which reproduces the original behavior on each of these trajectories.

It is straightforward to see that the quasi-static slicing includes both the static and dynamic slicing's. Indeed, when the set of variables V'_i is empty, quasi-static slicing reduces to static slicing, while for $V'_i = V_i$ a quasi-static slice coincides with a dynamic slice.

According to *De Lucia* in [Luc01], the notion of quasi static slicing is closely related to *partial evaluation* or *mixed computation* [BJE88], a technique to specialize programs with respect to partial inputs. By specifying the values of some of the input variables, constant propagation and simplification can be used to reduce expressions to constants. In this way, the values of some program predicates can be evaluated, thus allowing the deletion of branches which are not executed on the particular partial input. Quasi static slices are computed on specialized programs.

As told above, the need for quasi-static slicing arises from applications where the value of some input variables is fixed while the behavior of the program must be analyzed when other input values vary.

Listing 4.5 shows a quasi-static slice of the program in Listing 4.1 on the slicing criterion $C = (I', 38, sum)$ where $I' = \langle (test, 0) \rangle$.

```

1 main() {
2     int a, test, n, i, possum, negsum, sum;
3     scanf("%d",&test); scanf("%d",&n);
4     i = 1;
5     possum = negsum = 0;
6     while (i <= n) {
7         scanf("%d",&a);
8         if (a > 0) {
9             possum += a;
10        }
11        else if (a < 0) {
12            negsum -= a;
13        }
14        i++;
15    }
16    if (possum >= negsum) {
17        sum = possum;
18    }
19    else { sum = negsum; }
20    printf("Sum: %d\n", sum);
21 }
```

Listing 4.5: A quasi-static slice of program in Listing 4.1

Hence the value of variable *test* is zero, the *else if* branch is excluded from static slice. All the other conditional branches stay as part of the final quasi-static slice.

4.1.5 Conditioned Slicing

Canfora *et al* presented the *conditioned slicing* in [CCL98]. A conditioned slice consists of a subset of program statements which preserves the behavior of the original program with respect to a slicing criterion for any set of program executions. The set of initial states of the program that characterize these executions is specified in terms of a first order logic formula on the input.

Definition 61. Let V_i be the set of input variables of a program P , and F be a first order logic formula on the variables in V_i . A conditioned slicing criterion of a program P is a triple $C = (F(V_i), p, V_s)$ where p is a statement in P and V_s is the subset of the variables in P which will be analyzed in the slice.

Definition 62. Let V_i be a set of input variables of a program P and $F(V_i)$ be a first order logic formula on the variables in V_i . A satisfaction for $F(V_i)$ is any partial input I to the program for the variables in V_i that satisfies the formula F . The satisfaction set $S(F(V_i))$ is the set of all possible satisfactions for $F(V_i)$.

If V'_i is a subset of the input variables of the program P and $F(V'_i)$ is a first order logic formula on the variables in V'_i , each completion $I \in S(F(V'_i))$ of $I' \in S(F(V_i))$, identifies a trajectory T . A conditioned slice is any subset of the program which reproduces the original behavior on each of these trajectories.

Definition 63. A conditioned slice of a program P on a conditioned slicing criterion $C = (F(V_i), p, V_s)$ is any syntactically correct and executable program P' such that: P' is obtained from P by deleting zero or more statements; whenever P halts, on input I , with state trajectory T , where $I \in C(I', V'_i)$, $I' \in S(F(V_i))$, V'_i is the set of input variables of P , and S is the satisfaction set, then P' also halts, on input I , with state trajectory T' , and $Proj_{(p, V_s)}(T) = Proj_{(p, V_s)}(T')$.

A conditioned slice can be computed by first simplifying the program with respect to the condition on the input (i.e., discarding infeasible paths with respect to the input condition) and then computing a slice on the reduced program. A symbolic executor [Kin76] can be used to compute the reduced program, also called conditioned program in [CCL94]. Although the identification of the infeasible paths of a conditioned program is in general an *undecidable* problem, in most cases implications between conditions

can be automatically evaluated by a theorem prover. In [CCL98] conditioned slices are interactively computed: the software engineer is required to make decisions that the symbolic executor cannot make.

Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyze code fragments with respect to different perspectives.

Actually, conditioned slicing is a framework of statement deleting³ based methods, this is, the conditioned slicing criterion can be specified to obtain any form of slice.

Listing 4.6 shows a conditioned slice of the program in Listing 4.1 on the slicing criterion $C = (F(V_i), 38, \text{sum})$ where $V_i = \{n\} \cup_{1 \leq i \leq n} \{a_i\}$ and $F(V_i) = \forall i, 1 \leq i \leq n, a_i > 0$. The condition F imposes that all input values for the variable a are positive. This allows to be discard from the static slice of Listing 9.6 all the statements dependent on the condition $a < 0$ or $a == 0$.

```

1 main() {
2     int a, test, n, i, possum, negsum, sum;
3     scanf("%d",&test); scanf("%d",&n);
4     i = 1;
5     possum = negsum = 0;
6     while (i <= n) {
7         scanf("%d",&a);
8         if (a > 0) {
9             possum += a;
10        }
11        i++;
12    }
13    if (possum >= negsum) {
14        sum = possum;
15    }
16    printf("Sum: %d\n", sum);
17 }

```

Listing 4.6: A conditioned slice of program in Listing 4.1

4.1.6 Simultaneous Dynamic Slicing

Hall proposed the *simultaneous dynamic slicing* in [Hal95], which computes slices with respect to a set of program executions. This slicing method is called *simultaneous dynamic program slicing* because it extends dynamic slicing and simultaneously applies it to a set of test cases, rather than just one test case.

Definition 64. Let $\{T_1, T_2, \dots, T_k\}$ be a set of trajectories of length l_1, l_2, l_k , respectively, of a program P on input $\{I_1, I_2, \dots, I_k\}$. A simultaneous dynamic slicing criterion of P executed on each of the input I_j , $1 \leq j \leq k$, is a triple $C = (\{I_1, I_2, \dots, I_k\}, p, V_s)$ where p is a statement in P and V_s is a subset of the variables in P .

³Statement deletion means deleting a statement or a control predicate from a program.

Definition 65. A simultaneous dynamic slice of a program P on a simultaneous dynamic slicing criterion $C = (\{I_1, I_2, \dots, I_k\}, p, V)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I_j , $1 \leq j \leq m$, with state trajectory T_j , then P' also halts, on input I_j , with state trajectory T'_j , and $Proj_{(p, V_s)}(T_j) = Proj_{(p, V_s)}(T'_j)$.

A simultaneous program slice on a set of tests is not simply given by the union of the dynamic slices on the component test cases.

In [Hal95], Hall proposed an iterative algorithm that, starting from an initial set of statements, incrementally builds the simultaneous dynamic slice, by computing at each iteration a larger dynamic slice.

Listing 4.7 shows a simultaneous dynamic slice of the program in Listing 4.1 on the slicing criterion $C = (\{I_1, I_2\}, 38, sum)$ where

$$I_1 = \langle (test, 0), (n, 2), (a_1, 0), (a_2, 0) \rangle$$

and

$$I_2 = \langle (test, 1), (n, 2), (a_1, 0), (a_2, 2) \rangle$$

```

1 main() {
2     int a, test, n, i, possum, negsum, sum;
3     scanf("%d", &test); scanf("%d", &n);
4     i = 1;
5     possum = negsum = 0;
6     while (i <= n) {
7         scanf("%d", &a);
8         if (a > 0) {
9             possum += a;
10        }
11        else if (a < 0) {}
12        else if (test) {
13            if (possum >= negsum) {
14                possum = 0;
15            }
16        }
17        i++;
18    }
19    if (possum >= negsum) {
20        sum = possum;
21    }
22    printf("Sum: %d\n", sum);
23 }
```

Listing 4.7: A simultaneous dynamic slice of program in Listing 4.1

4.1.7 Union Slicing

Beszedes et al [BFS⁺02, BG02] introduced the concept of *union slice* and the computing algorithm. A union slice is the union of dynamic slices for a finite set of test cases; actually is very similar to simultaneous dynamic program slicing. A union slice is an approximation of a static slice and is much smaller than the static one.

The *union slicing criterion* is the same as the considered in the simultaneous dynamic slicing.

Definition 66. An union slice of a program P with different executions using the inputs $X = \{I_1, I_2, \dots, I_n\}$, with respect to a slicing criterion $C = (X, p, V_s)$, is defined as follows:

$$\text{UnionSlice}(X, p, V_s) = \bigcup_{I_k \in X} \text{DynSlice}(I_k, p, V_s)$$

where $\text{DynSlice}(I_k, i, V_s)$ contains those statements that influenced the values of the variables in V at the specific statement p .

Combined with static slices, the union slices can help to reduce the size of program parts that need to be investigated by concentrating on the most important parts first. The authors performed a series of experiments with three medium size C programs. The results suggest that union slices are in most cases far smaller than the static slices, and that the growth rate of union slices (by adding more test cases) significantly declines after several representative executions of the program. Thus, union slices are useful in software maintenance.

Daninic et al [DLH04] presented an algorithm for computing executable union slices, using conditioned slicing. The work showed that the executable union slices are not only applicable for program comprehension, but also for component reuse guided by software testing.

De Lucia et al [LHHK03] studied the properties of unions of slices and found that the union of two static slices is not necessarily a valid slice, based on Weiser's definition of a static slice. They argue that a way to get valid union slices is to propose algorithms that take into account simultaneously the execution traces of the slicing criteria, as in the simultaneous dynamic slicing algorithm proposed by Hall [Hal95].

Listing 4.8 shows an union slice of the program in Listing 4.1 on the slicing criterion $C = (I_1 \cup I_2, 38, \text{sum})$ where $I_1 = \langle (test, 0), (n, 2), (a_1, 0), (a_2, 2) \rangle$ and $I_2 = \langle (test, 0), (n, 3), (a_1, 1), (a_2, 0), (a_3, -1) \rangle$.

```

1 main() {
2     int a, test, n, i, possum, negsum, sum;
3     scanf("%d", &test); scanf("%d", &n);
4     i = 1;
5     possum = negsum = 0;
6     while (i <= n) {
7         scanf("%d", &a);
8         if (a > 0) {
9             possum += a;
10        }
11        else if (a < 0) {
12            negsum -= a;
13        }
14        i++;
15    }
16    if (possum >= negsum) {
17        sum = possum;
18    }
19    printf("Sum: %d\n", sum);
20 }
```

Listing 4.8: An union slice of program in Listing 4.1

4.1.8 Other Concepts

There are a number of other related approaches that use different definitions of slicing to compute subsets of program statements that exhibit a particular behavior. All these approaches add information to the slicing criterion to reduce the size of the computed slices.

Constrained Slicing

Field et al introduce in [FRT95] the concept of **constrained slice** to indicate slices that can be computed with respect to any set of constraints. Their approach is based on an intermediate representation for imperative programs, named PIM, and exploits graph rewriting techniques based on dynamic dependency tracking that model symbolic execution. The slices extracted are not executable. The authors are interested in the semantic aspect of more complex program transformations rather than in simple statement deletion.

Amorphous Slicing

Harman et al introduced **amorphous slicing** in [HBD03]. Amorphous slicing removes the limitation of *statement deletion* as the only means of simplification. Like a traditional slice, an amorphous program serves a projection of the semantics of the original program from which it is constructed. However, it can be computed by applying a broader range of transformation rules, including statement deletion.

Hybrid Slicing

Gupta et al presented the **hybrid slicing** in [GSH97], which incorporate both static and dynamic information. They proposed a slicing technique that exploits information readily available during debugging when computing slices statically.

4.1.9 Dicing

Dicing was a concept first introduced by *Lyle et al* in [LW86]. A **program dice** is defined as the set difference between the static slices of an incorrect variable and that of a correct variable, this is, the set of statements that potentially affect the computation of incorrect variable while do not affect the computation of the correct one. It is a fault location technique for further reducing the number of statements that need to be examined when debugging.

Later, in 1993, *Chen et al* [CC93], have proposed the dynamic program dicing.

4.1.10 Chopping

Chopping, introduced by Jackson [JR94], is a generalization of slicing. Although expressible as a combination of intersections and unions of *forward* and *backward* slices, *chopping* seems to be a fairly natural notion in its own right.

Two sets of instances form the criterion: source, a set of definitions, and sink, a set of uses. Chopping a program identifies a subset of its statements that account for all influences of the source on the sink. A conventional backward slice is a chop in which all the sink instances belong to the same site, and the source set contains every variable at every site. A chop is confined to a single procedure. The instances in source and sink must be within the procedure, and chopping only identifies statements in the text of the procedure itself.

After a survey of all the variants of the original slicing concept (static slicing) and its most important variant — the *dynamic slicing* — will be detailed in section 4.2 and 4.3 the original approach.

4.1.11 Relationships among Program Slicing Models

The slicing models discussed in the previous section can be classified according to a partial ordering relation, called *subsume relation*, based on the sets of program inputs specified by the slicing criteria. Indeed, for each of these slicing models, a slice preserves the behavior of the original program on all the trajectories identified by the set of program inputs specified by the slicing criterion.

In 1998, Canfora et al [CCL98] presented the concept of **subsume relation**.

Definition 67. *A program slicing model SM_1 subsumes a program slicing model SM_2 if for each slicing criterion defined according to SM_2 there exists an equivalent slicing criterion defined according to SM_1 that specifies the same set of program inputs.*

A slicing model is **stronger** than the slicing models it subsumes, because it is able to specify and compute slices with respect to a broader set of slicing criteria. Consequently, any slice computed according to a slicing model can also be computed with a stronger model. The subsume relation defines an hierarchy on the statement deletion based.

According to their definition of subsumes relation, the conditioned slicing subsumes any other model. Figure 4.1 shows the subsume hierarchy.

It is argued that the set of slicing models (static, dynamic, quasi-static, simultaneous, conditioned) is partially ordered with respect to subsume relation:

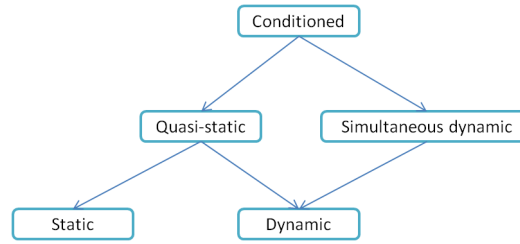


Figure 4.1: Relationships between program slicing models

- Quasi-static slicing subsumes static slicing;
- Quasi-static slicing subsumes dynamic slicing;
- Simultaneous dynamic slicing subsumes dynamic slicing;
- Conditioned slicing subsumes quasi-static;
- Conditioned slicing subsumes simultaneous dynamic slicing; and
- There is no relation between static slicing and dynamic slicing and between quasi-static slicing and simultaneous dynamic slicing.

In an attempt to formalize slicing concepts, Binkley et al defined a subsume relation in terms of syntactic ordering and semantic equivalence [BDG⁺04a, BDG⁺06]. This formalization establish a precise relationship between various forms of dynamic slicing and static slicing, counteracting the Canfora affirmation that there is no relation between static and dynamic slicing.

4.1.12 Methods for Program Slicing

According to Tip [Tip95] classification, there are three major kinds of approaches in program slicing:

- Dataflow equations;
- Information-flow relations; and
- Dependency graph based approaches.

The Weiser's original approach is a kind of method based on iteration of **dataflow equations**. In this approach, slices are computed in an iterative process, by computing consecutive sets of relevant statements for each node in the CFG. The algorithm first computes directly relevant statements for each node in the CFG, and then indirectly relevant statements are gradually added to the slice. The process stops when no more relevant statements are found.

Information flow relations for programs presented by Bergeretti [BC85] can also be used to compute slices. In this kind of approach, several types of relations are defined and computed in a syntax-directed, bottom-up manner. With these information-flow relations, slices can be obtained by relational calculus.

The most popular kind of slicing approach, the **dependency graph** approach, was proposed by Ottenstein and Ottenstein [OO84] and restate the problem of static slicing in terms of a reachability problem in a PDG. A slicing criterion is identified with a vertex in the PDG, and a slice correspond to all PDG vertices from which the vertex under consideration can be reached. Usually, the data dependencies used in program slicing are flow dependencies corresponding to the *DEF* and *REF* sets defined in section 2.1.

In these approaches using PDG, slicing can be divided into two steps. In the first step, the dependency graph of the program are constructed, and then the algorithm produce slices by doing graph reachability analysis over it.

As defined in Section 2.1, a dependency graph is a directed graph using vertexes to represent program statements and edges to represent dependencies. So, the graph reachability analysis can be done by traversing edges on the dependency graph from a node representing the slicing criteria. A dependency graph can represent not only dependencies but also other relations such as process communications and so on. Different slices can be obtained by constructing different dependency graphs.

4.2 Static Slicing

In this section it is discussed the basic static slicing approaches. Each subsection is divided according to the methods presented in previous section.

4.2.1 Basic Slicing Algorithms

In this subsection, are presented algorithms for static slicing of structured programs without non-scalar variables, procedures and interprocess communication.

Dataflow Equations

The original concept of program slicing [Wei81] was first proposed as the iterative solution to a dataflow problem specified using the program's control flow graph (CFG).

Definition 68. *A slice is statement-minimal if no other slice for the same criterion contains fewer statements.*

Weiser argues that statement-minimal slices are not necessarily unique, and that the problem of determining statement-minimal slices is undecidable.

Many researchers have investigated this problem, and various approaches result in good approximations. Some techniques are based on data-flow equations [KL88, LR87, Wei81] while others use graph representations of the program [AH90, Agr94, BH93a, Bin93, CF94, HRB88, JR94].

An approximation of statement-minimal slices are computed in an iterative process [Tip95], by computing consecutive sets of relevant variables for each node in the CFG. First, the directly relevant variables are determined, by only taking data dependencies into account. Below, the notation $i \rightarrow_{\text{CFG}} j$ indicates the existence of an edge in the CFG from node i to node j . For a slicing criterion $C = (n, V)$ (where n denotes the number line), the set of directly relevant variables at node i of the CFG, $R_C^0(i)$ is defined as follows:

- $R_C^0(i) = V$, when $i = n$;
- For every $i \rightarrow_C FGj$, $R_C^0(i)$ contains all variables v such that either
 - $v \in R_C^0(j)$ and $v \notin \text{DEF}(i)$;
 - $v \in \text{REF}(i)$ and $\text{DEF}(i) \cap R_C^0(j) \neq \emptyset$.

From this, a set of *directly relevant statements*, S_C^0 , is derived. S_C^0 is defined as the set of all nodes i which define a variable v that is relevant at a successor of i in the CFG:

$$S_C^0 = \{i \mid \text{DEF}(i) \cap R_C^0(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

Information-flow Relations

Bergeretti and Carré, in [BC85], proposed another approach that defines slices in terms of information-flow relations derived from a program in a syntax-directed fashion. The authors have defined a set of information-flow relations for sequences of statements, conditional statements and loop statements.

Dependency Graph based Approach

It were Karl Ottenstein and Linda Ottenstein [OO84] the first of many to define slicing as a reachability problem in a dependency graph representation of a program. They use the PDG for static slicing of single-procedure programs. The statements and expressions of a program constitute the vertices of a PDG, and edges correspond to data dependencies and control dependencies between statements (section 2.1). The key issue is that the partial

ordering of the vertices induced by the dependency edges must be obeyed so as to preserve the semantics of the program.

In the PDG's of Horwitz et al [HPR88, HRB88] a distinction is made between loop-carried and loop-independent flow dependencies. It is argued that the PDG variant of [HPR88] is minimal in the sense that removing any of the dependency edges, or disregarding the distinction between loop-carried and loop-independent flow edges would result in inequivalent programs having isomorphic PDGs.

The PDG variant considered in [OO84] shows considerably more detail than that of [HRB88]. In particular, there is a vertex for each (sub)expression in the program, and file descriptors appear explicitly as well.

In all dependency graph based approaches, the slicing criterion is identified with a vertex v in the PDG.

4.2.2 Slicing Programs with Arbitrary Control Flow

Dataflow Equations

In intraprocedural program slicing, the critical problem is to determine which predicates to be included in the slice when the program contains jump statements.

Lyle reports in [Lyl84] that the original slicing algorithm proposed by Weiser was able to determine which predicates to be included in the slice even when the program contains jump statements. It did not, however, make any attempt to determine the relevant jump statements themselves to be included in the slice. Thus, Weiser's algorithm may yield incorrect slices in the presence of unstructured control flow. Lyle presents a conservative solution for dealing with `goto` statements. His algorithm produces slices including every `goto` statement that has a non-empty set of active variables associated with it.

Gallagher [Gal90, GL91b] also use a variation of Weiser's method. In the algorithm, a `goto` statement is included in the slice if it jumps to a label of an included statement.

Jian, Zhou and Robson [JZR91] have also proposed a set of rules to determine which jump statements to include in a slice.

Agrawal shows in [Agr94] that either Gallagher algorithm or Jian et al algorithm does not produce correct slices in all cases.

Dependency Graph based Approach

Ball and Horwitz [BH93a, Bal93] and Choi and Ferrante [CF94] discovered independently that conventional PDG-based slicing algorithms produce incorrect results in the presence of unstructured control flow: slices may compute values at the criterion that differ from what the original program

does. These problems are due to the fact that the algorithms do not determine correctly when unconditional jumps such as `break`, `goto`, and `continue` statements are required in a slice. They proposed two similar algorithms to determine the relevant jump statements to include in a slice. Both of them require that jumps be represented as pseudo-predicates and the control dependency graph of a program be constructed from an augmented flow graph of the program. However, Choi and Ferrante distinguish two disadvantages of the slicing approach based on augmented PDGs (APDG). First, APDGs requires more space than conventional PDGs and their construction takes more time. Second, non-executable control dependency edges gives rise to spurious dependencies in some cases.

In their second approach, Choi and Ferrante also proposed another algorithm to construct an executable slice in the presence of jump statements when a “slice” is not constrained to be a subprogram of the original one. The algorithm constructs new jump statements to add to the slice to ensure that other statement in it are executed in the correct order.

The main difference between the approach by Ball and Horwitz and the first approach of Choi and Ferrante is that the latter use a slightly more limited example language: conditional and unconditional `goto`’s are present, but no structured control flow constructs. Although Choi and Ferrante argue that these constructs can be transformed into conditional and unconditional `goto`’s. Ball and Horwitz show that, for certain cases, this results in overly large slices.

Both groups have been proposed two formal proofs to show that their algorithms compute correct slices.

Agrawal [Agr94] proposed an algorithm has the same precision as that of the above two algorithms. He observes that a conditional jump statement of the form `if P then goto L` must be included in the slice if the predicate `P` is in the slice because another statement in the slice is control dependent on it. This algorithm is appealing in that it leaves the flow-graph and the PDG of the program intact and uses a separate graph to store the additional required information. It lends itself to substantial simplification, when the program under consideration is a structured program. Also, the simplified algorithm directly leads to a conservative approximation algorithm that permits on-the-fly detection of the relevant jump statements while applying the conventional slicing algorithms.

Harman and Danicic [HD98] defined an extension of Agrawal’s algorithm that produces smaller slices by using a refined criterion for adding jump statements (from the original program) to the slice computed using Ottenstein’s algorithm for building and slicing the PDG [OO84].

Kumar and Horwitz [KH02] extended the previous work on program slicing by providing a new definition of “correct” slices, by introducing a representation for C-style `switch` statements, and by defining a new way to compute control dependencies and to slice a PDG so as to compute more

precise slices of programs that include jumps and switches.

4.2.3 Interprocedural Slicing Methods

Dataflow Equations

Weiser describes a two-step approach for computing interprocedural static slices in [Wei81]. In the first step, a slice is computed for the procedure P which contains the original slicing criterion. The effect of a procedure call on the set of relevant variables is approximated using interprocedural summary information [Bar78]. For a procedure P , this information consists of a set $MOD(P)$ of variables that may be modified by P , and a set of $USE(P)$ of variables that may be used by P , taken into account any procedures called by P . The fact of Weiser's algorithm does not take into account which output parameters are dependent on which input parameters is a cause of imprecision. This is illustrated in program 4.9 listed below. The Weiser's interprocedural slicing algorithm will compute the slice listed in program 4.10. This slice contains the statement `int a = 17;` due to the spurious dependency between variable a before the call, and variable d after the call.

```

2 void simpleAssign(int v, int w, int x, int y) {
    x = v;
    y = w;
4 }
6 main() {
    int a = 17;
8    int b = 18;
    int c, d;
10   simpleAssign(a,b,c,d);
    printf("Result: %d\n", d);
12 }
```

Listing 4.9: Interprocedural sample program

```

2 void simpleAssign(int v, int w, int x, int y) {
    y = w;
4 }
6 main() {
    int a = 17;
8    int b = 18;
    int c, d;
10   simpleAssign(a,b,c,d);
}
```

Listing 4.10: Weiser's Interprocedural slice of program 4.9

In the second step of Weiser's algorithm new criteria are generated for:

- a) Procedures Q called by P ;
- b) Procedures R that call P .

The two steps described above are repeated until no new criteria occur. The criteria of a) consists of all pairs (n_Q, V_Q) where n_Q is the last statement of Q and V_Q is the set of relevant variables in P which is in the scope of

Q (where formals are substituted by actual). The criteria of b) consists of all pairs (n_R, V_R) such that N_R is a call to P in R , and V_R is the set of relevant variables at the first statement of P which is in the scope of R (actuals are substituted by formals). The generation of new criteria is formalized by way of functions $UP(\mathcal{S})$ and $DOWN(\mathcal{S})$ which map a set \mathcal{S} of slicing criteria in a procedure P to a set of criteria in procedures that call P , and a set of criteria in procedures called by P , respectively. The closure $UP \cup DOWN^*(\{C\})$ contains all criteria necessary to compute an interprocedural slice, given an initial criterion C . Worst-case assumptions have to be made when a program calls external procedures, and the source code is unavailable.

Horwitz, Reps and Binkley report that Weiser's algorithm for interprocedural slicing is unnecessarily inaccurate because of what they refer to as the "calling context" problem, i.e., the transitive closure operation fails to account for the calling context of a called procedure. In a nutshell, the problem is that when the computation 'descends' into a procedure Q that is called from a procedure P , not only P . This corresponds to execution paths which enter Q from P and exit Q to a different procedure P' .

Tip [Tip95] conjecture that the calling context problem of Weiser's algorithm can be fixed by observing that the criteria in the UP sets are only needed to include procedures that transitively call the procedure containing the initial criterion. Once this is done, only $DOWN$ sets need to be computed.

Hwang, Du and Chou [JDC88] proposed an iterative solution for interprocedural static slicing based on replacing recursive calls by instances of the procedure body. The slice is recomputed in each iteration until a fixed point is found (i.e., no new statement are added to a slice). This approach do not suffer from the calling context problem because expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps [RHSR94, Rep96] showed that for a certain family P^k of recursive programs, this algorithm takes time $O(2^k)$, i.e., exponential in the length of the program.

Dependency Graph based Approach

Interprocedural slicing as a graph reachability problem requires extending of the PDG and, unlike the addition of data types or unstructured control flow, it also requires modifying the slicing algorithm. The PDG modifications represents call statements, procedure entry, parameters, and parameter passing. The algorithm change is necessary to correctly account for procedure calling context.

Horwitz, Reps and Binkley [HRB88] introduce the notion of *System Dependency Graph* (SDG) for the dependency graphs that represents multi-procedure programs. Figure 4.2 shows the SDG corresponding to the pro-

gram 4.11 listed below.

```

2  void Add(int* a, int b) {
   *a = *a + b;
   }
4
6  void Increment(int z) {
   Add(&z, 1);
   }
8
10 void A(int x, int y) {
   Add(&x, y);
   Increment(&y);
12 }
14 void main() {
   int sum = 0;
16   int i = 1;
   while (i < 11) {
18     A(&sum, i);
   }
20   printf("Sum: %d\n", sum);
   }

```

Listing 4.11: Program Example

Interprocedural slicing can be defined as a reachability problem using the SDG, just as intraprocedural slicing is defined as a reachability problem using the PDG. The slices obtained using this approach are the same as those obtained using Weiser’s interprocedural slicing method [Wei84]. However, this approach does not produce slices that are as precise as possible, because it considers paths in the graph that are not possible execution paths. For example, there is a path in the SDG shown in Figure 4.2 from the vertex of procedure *main* labeled “ $x_{in} = \text{sum}$ ” to the vertex of *main* labeled “ $i = y_{out}$ ”. However this path corresponds to procedure *Add* being called by procedure *A*, returning to procedure *Increment*, which is not possible. The value of *i* after the call to procedure *A* is independent of the value of *sum* before the call, and so the vertex labeled “ $x_{in} = \text{sum}$ ” should not be included in the slice with respect to the vertex labeled “ $i = y_{out}$ ”. Figure 4.3 shows this slice.

To achieve more precise interprocedural slices, an interprocedural slice with respect to vertex *s* is computed using two passes over the graph. Summary edges permit moving across a call procedure; thus, there is no need to keep track of calling context explicitly to ensure that only legal execution paths are traversed. Both passes operate on the SDG, traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges. Informally, if *s* is in procedure *P* then pass 1 identifies vertices that reach *s* and are either in *P* itself or procedures that (transitively) call *P* [BG96]. The traversal in pass 1 does not descend into procedures called by *P* or its callers. Pass 2 identifies vertices in called procedures that induce the summary edges used to move across call sites in pass 1.

The traversal in pass 1 starts from *s* and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. The traversal in

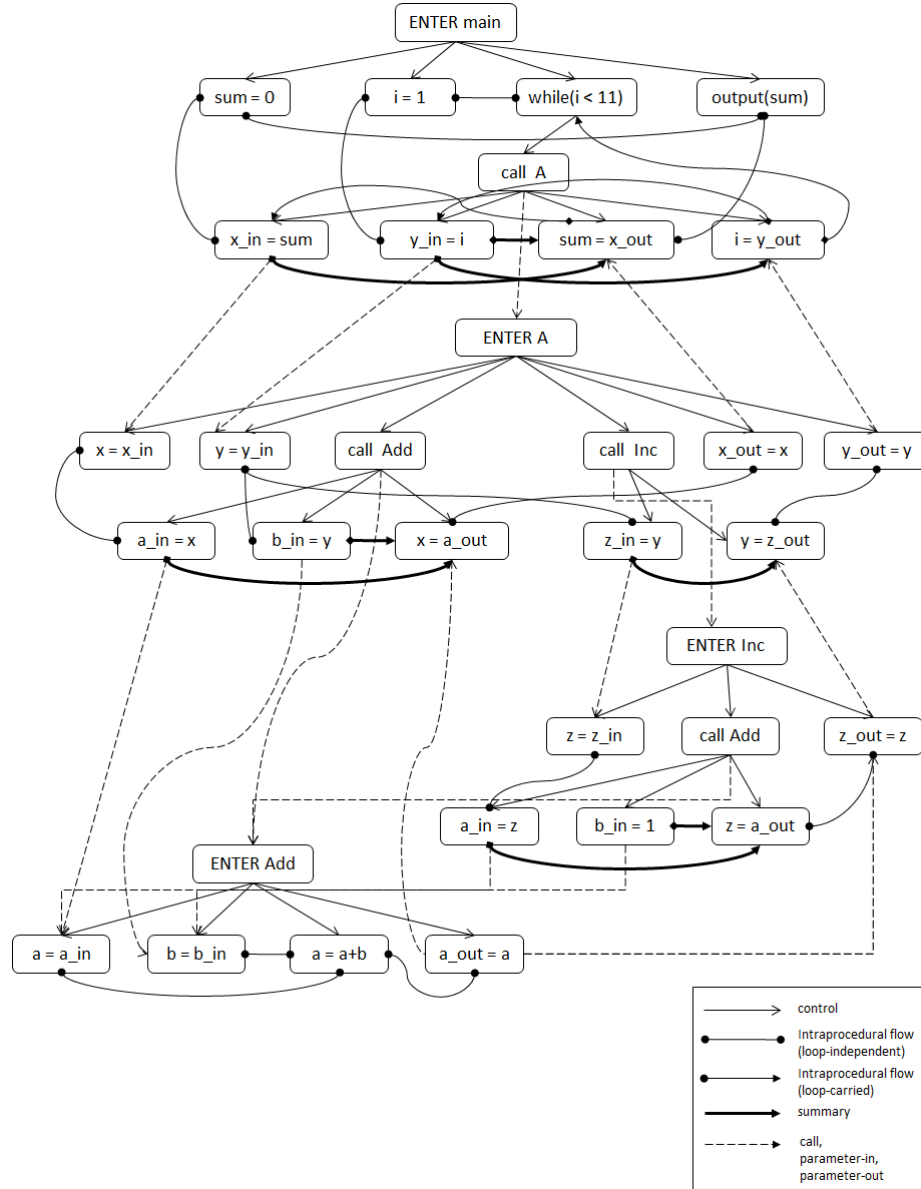


Figure 4.2: Example system and its SDG

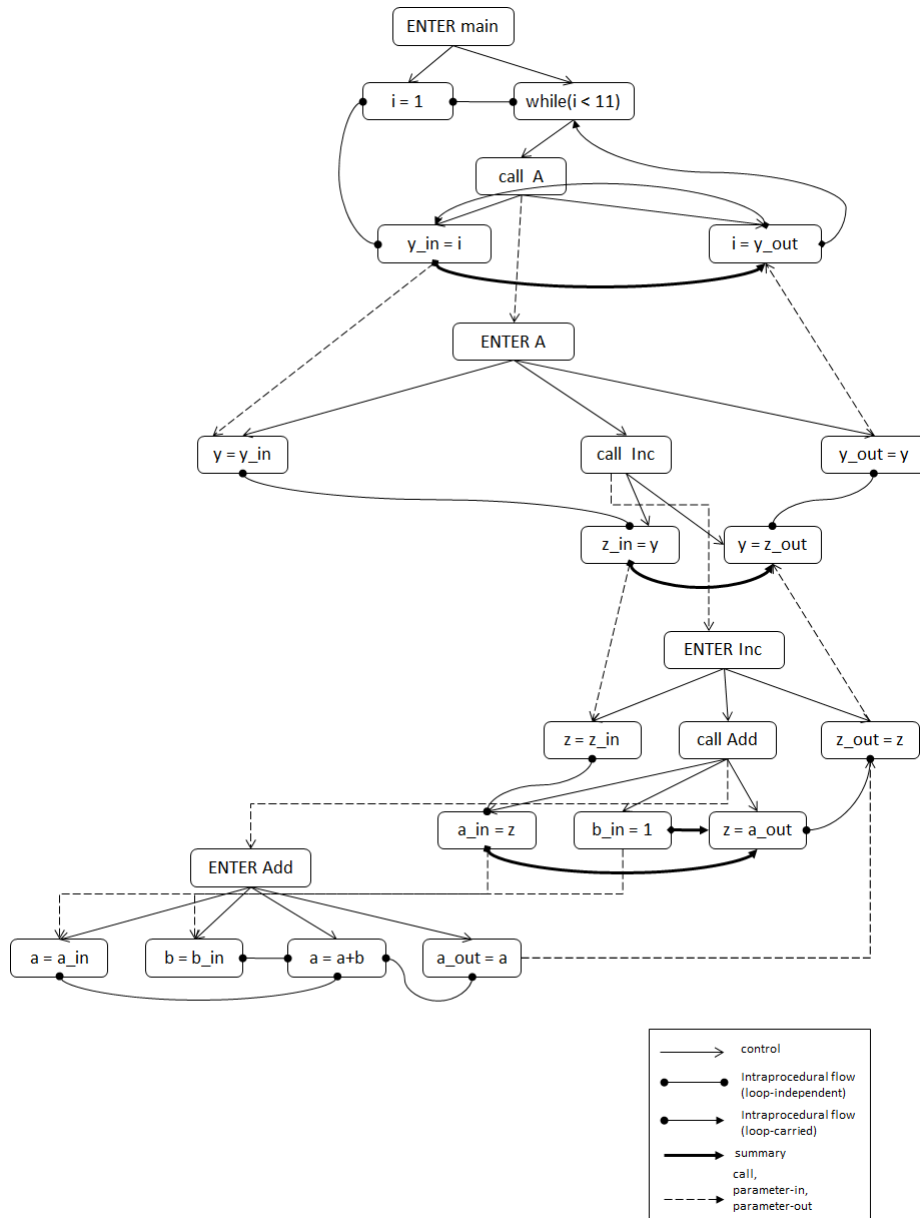


Figure 4.3: The SDG from Figure 4.2 sliced with respect to the formal-out vertex for parameter z in procedure *Increment*, together with the system to which it corresponds.

pass 2 starts from all vertices reached in pass 1 and goes backwards along flow edges, control edges, summary edges, and parameter-out edges, but not along call, or parameter-in edges. The result of an interprocedural slice of a graph G with respect to vertex set S , denoted by $Slice(G, S)$, consists of the sets of vertices encountered during by pass 1 and pass 2, and the set of edges induce by this vertex set.

$Slice(G, S)$ is a subgraph of G . However it may be infeasible (i.e., it may be not the SDG of any system). The problem arises when $Slice(G, S)$ includes mismatched parameters: different call-sites on a procedure include different parameters. There are two causes of mismatches: missing actual-in vertices and missing actual-out vertices. Making such systems syntactically legal by simply adding missing parameters leaves semantically unsatisfactory systems [Bin93]. In order to include the program components necessary to compute a safe value for the parameter represented at missing actual-in vertex v , the vertices in the pass 2 slice of G taken with respect to v must be added to the original slice. A pass 2 slice includes the minimal number of components necessary to produce a semantically correct system. The addition of pass 2 slices is repeated until no further actual-in vertex mismatches exist.

The second cause of parameter mismatches is missing actual-out vertices. Because missing actual-out vertices represent dead-code no additional slicing is necessary. Actual-out mismatches are removed by simply adding missing actual-out vertices to the slice.

The details of this algorithm are given in [Bin93].

Several extensions of Horwitz-Reps-Binkley (HRB) algorithm have been presented. Lakhotia [Lak92] adapted the idea of lattice theory to interprocedural slicing and presented a slicing algorithm based on the augmented SDG in which a tag is contained for each vertex of SDG. Different from HRB algorithm, this one only need one traverse on the SDG. Binkley extended HRB algorithm to produce executable interprocedural program slices in [Bin93].

Clarke et al [CFR⁺99] extended HRB algorithm to VHDL (Very High Speed Integrated Circuit Hardware Description Language), using an approach based on capturing the operational semantics of VHDL in traditional constructs. Their algorithm first maps the VHDL constructs onto traditional program language constructs and then slices using a language-independent approach.

Orso et al [OSH01] proposed a SDG-based incremental slicing technique, in which slices are computed based on types of data dependencies. They classified the data dependencies into different types. The scope of a slice can be increased in steps, by incorporating additional types of data dependencies at each steps.

To the problem of SDG constructing, Forgacsy and Gyimóthy [FG97] presented a method to reduce the SDG. Livadas and Croll [LC94] extended the SDG and proposed a method to construct SDG directly from parser

trees. Their algorithm is conceptually much simpler, but it cannot handle recursion. Kiss [KJLG03] presented an approach to construct SDG from the binary executable programs and proposed an algorithm to slice on them. Sinha, Harrold and Rothermel [SHR99] extended the SDG to represent interprocedural control dependencies. Their extension is based on Augmented Control Flow Graph (ACFG), a CFG augmented with edges to represent interprocedural control dependencies. Hisley et al [HBP02] extended the SDG to threaded System Dependency Graph (tSDG) in order to represent non-sequential programs.

Information-flow Relations

Bergeretti and Carré explains in [BC85] how the effect of procedure calls can be approximated. Exact dependencies between input and output parameters are determined by slicing the called procedure with respect to which output parameter (i.e., the computation of the μ relation for the procedure). Then, each procedure call is replaced by a set of assignments, where each output parameter is assigned to a fictitious expression that contains the input parameters it depends on. As only feasible execution paths are considered, this approach does not suffer from the calling context problem. A call to a side-effect free function can be modeled by replacing it with a fictitious expression containing all actual parameters. Note that the computed slices are not truly interprocedural since no attempt is done to slice procedures other than the main program.

4.2.4 Slicing in the Presence of Composite Datatypes and Pointers

Dependency Graph based Approach

When slicing, there are two approaches to handle arrays. A simple approach for arrays is to treat each array as a whole [Lyl84]. According to Lyle, any update to an element of an array is regarded as an update and a reference of the entire array. However, this approach leads to unnecessary large slices. To be more precise requires distinguishing the elements of array. And this needs dependency analysis.

The PDG variant of Ottenstein and Ottenstein [OO84] contains a vertex for each sub-expression; special *select* and *update* operators serve to access elements of an array.

Banerjee [Ban88] presented the Extended GDC Test. It can be applied to analyze the general objects (multi-dimensional arrays and nested trapezoidal loops). The test is derived from number theory. The single equation $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$ has an integer solution if and only if $gcd(a_i)$ divides b . This give us an exact test for single-dimensional arrays ignoring

bounds. And it can be extended to multi-dimensional arrays.

In the presence of pointers, situations may occur where two or more variables refer to the same memory location. This phenomenon is commonly called *aliasing*. Algorithms for determining potential aliases can be found in [CBC93, LR92]. Slicing in the presence of aliasing requires a generalization of the notion of data dependency to take potential aliases into account. Tip [Tip95] have presented the definition of *potentially data dependent*.

Definition 69. *A statement s is potentially data dependent on a statement s' if:*

- i) s defines a variable X' ;
- ii) s' uses a variable X ;
- iii) X and X' are potential aliases; and
- iv) there exists a path from s to s' in the CFG where X is not necessarily defined.

Such paths may contain definitions to potential aliases of X .

A slightly different approach is pursued by Horwitz et al in [HPR89a]. Instead of defining data dependency in terms of potential definitions and uses of variables, they defined this notion in terms of potential definitions and uses of *abstract memory locations*.

However, Landi [Lan92] have shown that precise pointer analysis is undecidable. So the analysis has to do a trade-offs between cost and precision. There are several dimensions that affect the trade-offs. How a pointer analysis addresses each of these dimensions helps to categorize the analysis.

Besides the data dependency, in the presence of pointers, the reaching definition also need to be changed, and the *l-valued* expression have to be taken into account.

Definition 70. *An l-valued expression is any expression which may occur as the left-hand side of an assignment.*

Jiang [JZR91] presented an algorithm for slicing C programs with pointers and arrays. Unfortunately, the approach appears to be flawed. There are statements incorrectly omitted, resulting in inaccurate slices.

4.3 Dynamic Slicing

In this section is is discussed the dynamic slicing approaches.

4.3.1 Basic Algorithms for Dynamic Slicing

Dataflow Equations

It was Korel and Laski [KL88, KL90] who first proposed the notion of dynamic slicing. As it was defined in section 4.1.3, a dynamic slice is a part of a program that affects the concerned variable in a particular program execution. As only one execution is taken into account, dynamic program slicing may significantly reduce the size of the slice as compared to static slicing.

Most dynamic slices are computed with respect to an *execution history* or *trajectory*. This history records the execution of statements as the program executes. The execution of a statement produces an occurrence of the statement in the trajectory. Thus, the trajectory is a list of statement occurrences.

Two example execution histories are shown below for the program 4.12. Superscripts are used to differentiate between the occurrences of a statement. For example, statement 2 executes twice for the second execution producing 2^1 and 2^2 .

```

1  scanf("%d", &n);
2  for (i = 1; i < n; i++) {
3      a = 2;
4      if (c1) {
5          if (c2) {
6              a = 4;
7          }
8          else {
9              a = 6;
10         }
11     }
12     z = a;
13     printf("Result: %d\n", z);

```

Listing 4.12: Two execution histories

Execution history 1

Input $n = 1$, $c1$ and $c2$ both true:

$\langle 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 12^1, 2^2, 14^1 \rangle$

Execution history 2

Input $n = 2$, $c1$ and $c2$ false on the first iteration and true on the second:

$\langle 1^1, 2^1, 3^1, 4^1, 12^1, 2^2, 3^2, 4^2, 5^1, 6^1, 12^2, 2^3, 14^1 \rangle$

In order to compute dynamic slices, Korel and Laski introduce three dynamic flow concepts which formalize the dependencies between occurrences of statements in a trajectory.

Definition 71. Definition Use (*DU*). $v^i \text{ DU } w^j \Leftrightarrow v^i$ appears before w^j in the execution history and there is a variable x defined at v^i , used at w^j , but not defined by any occurrence between v^i and w^j .

This definition captures flow dependency that arise when one occurrence represent the assignment of a variable and another use of that variable. For example, in program 4.12 listed above, when $c1$ and $c2$ are both false, there is a flow dependency from statement 3 to statement 12.

Definition 72. Test Control (*TC*). $v^i \text{ TC } w^j \Leftrightarrow w^j$ is control dependent on v^i (in the static sense) and for all occurrences w^k between v^i and w^j , w^k is control dependent on v^i .

This second definition captures control dependency. The only difference between this definition and the static control dependency definition is that multiple occurrence of predicates exist.

Definition 73. Identity Relation (IR). $v^i IR u^j \Leftrightarrow v = u$.

Dynamic slices are computed in an iterative way, by determining successive sets S^i of directly and indirectly relevant statements. For a slicing criterion $C = (I^q, p, V)$ the initial approximation S^0 contains the last definition of the variables in V in the trajectory, as well as the test actions in the trajectory on which I^q is control dependent. Approximation S^{i+1} is defined as follows:

$$S^{i+1} = S^i \cup A^{i+1}$$

where A^{i+1} consists of:

$$A^{i+1} = X^p | X^p \notin S^i, (X^p, Y^t) \in (DU \cup TC \cup IR) \text{ for some } Y^t \in S^i, p < q$$

The dynamic slice is obtained from the fixpoint S_C of this process (as q is finite, this always exists): any statement X for an instance X^p occurs in S_C will be in the slice.

Program 4.13 shows the Korel and Laski slice of the program shown in program 4.12 taken with respect to $(3^2, 2, \{a\})$.

```

2   scanf("%d", &n);
   for (i = 1; i < n; i++) {
4       a = 2;
   }
```

Listing 4.13: A dynamic slice of the program listed in program 4.12 and its execution history

This slice is computed as follows:

$$\begin{aligned}
DU &= \{(1^1, 2^1), (3^1, 12^1), (6^1, 12^2), (12^2, 14^1)\} \\
TC &= \{(2^1, 3^1), (2^1, 4^1), (2^1, 12^1), (2^1, 3^2), (2^1, 4^2), (2^1, 12^2), \\
&\quad (2^2, 3^2), (2^2, 4^2), (2^2, 12^2), (4^2, 5^1), (5^1, 6^1)\} \\
IR &= \{(2^1, 2^2), (2^1, 2^3), (2^2, 2^3), (3^1, 3^2), (4^1, 4^2), (12^1, 12^2)\} \\
S^0 &= \{2^1\} \\
S^1 &= \{1^1, 2^1\} \\
S^2 &= \{1^1, 2^1\} = S^1, \text{ thus the iteration ends.} \\
S &= \{3^1\} \cup \{1^1, 2^1\} = \{1^1, 2^1, 3^1\}.
\end{aligned}$$

Information-flow Relations

Gopal [Gop91] have proposed an approach where *dynamic dependency relations* are used to compute dynamic slices. He introduces dynamic versions of Bergeretti and Carré information-flow relations. The $\overline{\lambda}_S$ relations contains all pairs (v, e) such that statement e depends on the input value of v when program S is executed. Relation $\overline{\mu}_S$ contains all pairs (e, v) such that the

output value of v depends on the execution of statement e . A pair (v, v') is in the relation $\overline{\rho_S}$ if the output value of v' depends on the input value of v . In this definitions it is presumed that S is executed for some fixed input.

For empty statements, assignments, and statement sequences Gopal's relations are exactly the same as for the static case.

Dependency Graph based Approach

Miller and Choi [MC88] first proposed the notion of dynamic dependency graph. However, their method mainly concentrates on parallel program debugging and flowback analysis⁴.

Agrawal and Horgan [AH90] developed an approach for using dependency graphs to compute non-executable dynamic slices. Their first two algorithms for computing dynamic slices are inaccurate. The initial approach uses the PDG and marks the vertices that are executed for a given test. A dynamic slice is computed by computing a static slice in the sub-graph of the PDG that is induced by the marked vertices. By construction, this slice only contains vertices that are executed. This solution is imprecise because it does not detect situations where there exists a flow edge in the PDG between a marked vertex v_1 and a marked vertex v_2 , but where the definitions of v_1 are not actually used at v_2 .

The second approach consists of marking PDG edges as the corresponding dependencies arise during execution. Again, the slice is obtained by traversing the PDG, but this time only along marked edges. Unfortunately, this approach still produces imprecise slices in the presence of loops because an edge that is marked in some loop iteration will be present in all subsequent iterations, even when the same dependency does not recur.

This approach computes imprecise slices because it does not account for the fact that different occurrences of a statement in the execution history may be (transitively) dependent on different statements. This observation motivate their third solution: create a distinct vertex in the dependency graph for each occurrence of a statement in the execution history. This kind of graph is referred as *Dynamic Dependency Graph* (DDG). A dynamic slicing criterion is identified with a vertex in the DDG, and a dynamic slice is computed by determining all DDG vertices from which the criterion can be reached.

A dynamic slice can now be defined in terms of a variable, an execution history, and an occurrence of a statement in the execution history. The slice contains only those statements whose execution had some effect on the value of the variable at the occurrence of the statement in the execution history.

⁴Flowback analysis is a powerful technique for debugging programs. It allows the programmer to examine dynamic dependencies in a program's execution history without having to re-execute the program.

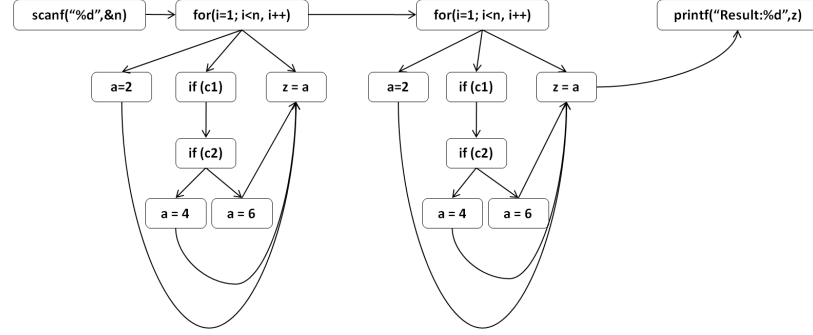


Figure 4.4: The DDG for the example listed in program 4.12.

Figure 4.4 shows the DDG for program 4.12, considering the slice on variable z at statement 14, where c_1 and c_2 false on the first iteration and true on the second.

Goswami and Mall [GM02] presented a dynamic algorithm based on the notion of compact dynamic dependency graph (CDDG). The control dependency edges of the CDDG are constructed statically while the data-flow dependency edges are constructed dynamically.

Mund et al [MMS03] found that CDDG-based approaches may not produce correct result in some cases. They proposed three intraprocedural dynamic slicing methods, two based on marked PDG and another based on their notion of *Unstructured Program Dependency Graph* (UPDG) which can be used for unstructured programs. Their first method also based on the marking and unmarking of edges, while the other two based on the runtime marking and unmarking of nodes. It is claimed that all the three algorithms are precise and more space and time efficient than former algorithms.

Zhang et Gupta [ZG04] found that different dynamic dependency could be expressed by one edge in the dependency graph. They presented a practical dynamic slicing algorithm which is based upon a novel representation of the dynamic dependency graph that is highly compact and rapidly traversable.

Further, Zhang et al [ZGZ04] studied the statistical characteristics of dynamic slices by experiments. Based on the forward slicing methods, they introduced a way of using reduced ordered binary decision diagrams (roBDDs) to represent a set of dynamic slices. Within this technique, the space and time requirements of maintaining dynamic slices are greatly reduced. Thus, the efficiency of dynamic slicing can be improved.

4.3.2 Slicing Programs with Arbitrary Control Flow

Dependency Graph based Approach

Korel [Kor97a, Kor97b] used a removable block based approach to handle jump statements in dynamic program slicing. This approach can produce correct slices in the presence of unstructured programs.

Huynh and Song [HS97] then extended the forward dynamic slicing method presented in [KY94] to handle jump statements. However, their algorithm can handle unstructured programs having only structured jumps.

Mund, Mall and Sarkar [MMS03] proposed a notion of jump dependency. Based on this notion, they build the *Unstructured Program Dependency Graph* (UPDG) as the intermediate representation of a program. Their slicing algorithm based on UPDG can produce precise slices.

Faragó and Gergely [FG02] handled jump statements for the forward dynamic slicing by building a transformed D/U structure for all relevant statements. This method can be applied to goto, break, continue and switch statements of C programs.

4.3.3 Interprocedural Slicing Methods

Dependency Graph based Approach

Several approaches have been presented concerning on interprocedural dynamic slicing.

In [ADS91], Agrawal et al consider dynamic slicing of procedures with various parameter-passing mechanisms. *Call-by-value* parameter-passing is modeled by a sequence of assignments $f_1 = a_1; \dots; f_n = a_n$;⁵ which is executed before the procedure is entered. In order to determine the memory cells for the correct activation record, the *USE* (see section 2.1) sets for the actual parameters a_i are determined before the procedure is entered, and the *DEF* sets for the formal parameters f_i after the procedure is entered.

For *Call-by-value-result* parameter passing, additional assignments of formal parameters to actual parameters have to be performed upon exit from the procedure.

Call-by-reference parameter-passing does not require any actions specific to dynamic slicing, as the same memory cell is associated with corresponding actual and formal parameters a_i and f_i .

Notice that in this approach dynamic data dependencies based on definitions and uses of memory location are used. This way, two potential problems are avoided. First, the use of global variables inside procedures does not pose any problems. Second, no alias analysis is required.

⁵It is assumed that a procedure P with formal parameters f_1, \dots, f_n is called with actual parameters a_1, \dots, a_n

Kamkar et al [KSF92, KFS93b] further discussed the problem of intraprocedural dynamic slicing. They proposed a method that primarily concerned with procedure level slices. That is, they study the problem of determining the set of *call sites* in a program that affect the value of a variable at a particular call site.

During execution, a *dynamic dependency summary graph* is constructed. The vertices of this graph, referred to as *procedure instances*, correspond to procedure activations annotated with their parameters. The edges of the summary graph are either activations edges corresponding to procedure calls, or summary dependency edges. The latter type reflects transitive data and control dependencies between input and output parameters of procedure instances.

A slicing criterion is defined as a pair consisting of a procedure instance, and an input or output parameter of the associated procedure. After constructing the summary graph, a slice with respect to a slicing criterion is determined in two steps. First, the parts of the summary graph from which the criterion can be reached is determined; this subgraph is referred to as an execution slice. Vertices of an execution slice are *partial* procedure instances, because some parameters may be “sliced away”. An interprocedural program slice consists of all call sites in the program for which a partial instance occurs in the execution slice.

4.3.4 Slicing in the Presence of Composite Datatypes and Pointers

Dataflow Equations

Korel and Laski [KL90] consider slicing in the presence of composite variables by regarding each element of an array, or field of a record as a distinct variable. Dynamic data structures are treated as *two* distinct entities, namely the pointer itself and the object being pointed to. For dynamically allocated objects, they propose a solution where a unique name is assigned to each object.

Dependency Graph based Approach

Agrawal et al [ADS91] present a dependency graph based algorithm for dynamic slicing in the presence of composite datatypes and pointers. Their solution consist of expressing *DEF* and *USE* sets in terms of *actual memory locations* provided by the compiler. The algorithm presented is similar to that for static slicing in the presence of composite datatypes and pointers by the same authors.

Faragó [FG02] also discussed the problem of handling pointers, arrays and structures for C programs when doing forward dynamic slicing. Abstract

memory locations are used in this method and program instrumentation is used to extract these locations.

4.4 Applications of Program Slicing

As discussed in the previous sections, program slicing is a well-recognized technique that is used mainly at source code level to highlight code statements that impact upon other statements. Slicing has many applications because it allows a program to be simplified by focusing attention on a sub-computation of interest for a chosen purpose. In this section we present some of the applications of program slicing.

4.4.1 Debugging

The original motivation for program slicing was to aid the location of faults during debugging activities. The idea was that the slice would contain the fault, but would not contain lines of code that could not have caused the failure observed. This is achieved by setting the slice criterion to the variable for which an incorrect value is observed.

Clearly slice cannot be used to identify bugs such as missing initialization of a variable. If the original program does not contain a line of code the slice will not contain it either. Although slicing cannot identify omission errors, Harman have argued that slicing can be used to aid the detection of such errors [HBD03].

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior. Dynamic slices are particular useful here because they only reflect the actual dependencies of that execution, resulting in smaller slices than static ones. In his thesis, Agrawal discussed how static and dynamic slicing can be used for semi-automated debugging of programs. He proposed an approach where the user gradually 'zooms out' from the location where the bug manifested itself by repeatedly considering larger data and control slices.

Slicing is also useful in algorithmic debugging [FSKG92]. An algorithm debugger partially automates the task of localizing a bug by comparing the intended program behavior with the actual program behavior. The intended behavior is obtained by asking the user whether or not a procedure (program unit) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level.

Debugging was also the motivation for program dicing and latter program chopping (see section 4.1.10). Dicing uses the information that some variables fail some tests, while other variables pass all tests, to automatically identify a set of statements likely to contain the bug [LW87]. The technique of program chopping identifies the statement that transmit values from a statement t to a statement s . A program chop is useful in debugging when

a change at t causes an incorrect result to be produced at s . The statements in $\text{chop}(t, s)$ are the statements that transmit the effect of the change at t to s . Debugging attention should be focused here. In the absence of procedures, $\text{chop}(t, s)$ is simply the intersection of the forward slice taken with respect to t and the backward slice taken with respect to s can be viewed as a generalized kind of program dice.

4.4.2 Software Maintenance

Software maintainers are faced with the upkeep of programs after their initial release and experiment the same problems as program integrators: understanding existing software and making changes without having a negative impact on the unchanged part.

Gallagher and Lyle [GL91b] use the notion of decomposition slice of programs. A decomposition slice with respect to a variable v consists of all statements that may affect the observable value of v at some point; it is defined as the union of slices with respect to v at any statement that outputs v , and the last statement of the program. Essentially, they decompose a program into a set of components (reduced programs), and each of them captures part of the original program's behavior. The main observation of [GL91b] is that independent statements in a slice do not affect the data and control flow in the complement. This results in the follow guidelines for modification:

- Independent statements may be deleted from a decomposition slice;
- Assignments to independent variables may be added anywhere in a decomposition slice;
- Logical expressions and output statements may be added anywhere in a decomposition slice;
- New control statements that surround any dependent statements will affect the complement's behavior.

Slicing can also be used to identify reusable functions [CLLF94, CCLL94, CLM95, CLM96, LV97]. Canfora et al presented a method to identify functional abstraction in existing code [CLLF94]. In this approach, program slicing is used to isolate the external functions of a system and these are then decomposed into more elementary components by intersection slices. They also found that conditioned slice could be used to extract procedures from program functionality.

4.4.3 Reverse Engineering

Besides above application in software maintenance, program slicing can be used in reverse engineering [BE93, JR94]. Reverse engineering concerns the

problem of comprehending the current design of a program and the way this design differs from the initial development. This involves abstracting out of the source code, the design decisions and rationale from the initial development (design recognition) and understanding algorithms chosen (algorithm recognition).

Beck and Eichmann [BE93] applied program slicing techniques to reverse engineering by using it to assist in the comprehension of large software systems, through traditional slicing techniques at the statement level, and through a new technique, *interface slicing*, at the module level. A dependency model for reverse engineering should treat procedures in a modular fashion and should be fine-grained, distinguishing dependencies that are due to different variables. Jackson and Rollins [JR94] proposed an improved PDG that satisfies both, while retaining the advantages of PDG. They proposed an algorithm to compute chopping from their dependency graph which can produce more accurate results than algorithms based directly on the PDG.

4.4.4 Program Comprehension

Program comprehension is a vital software engineering and maintenance activity. It is necessary to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems.

Slicing revealed to be helpful in the comprehension phase of maintenance. De Lucia et al [LFM96] used conditioned slicing to facilitate program comprehension. Quasi-static slicing can also be used in program comprehension. These techniques share the property that a slice is constructed with respect to a condition in addition to the traditional static slicing and thus can give the maintainer the possibility to analyze code fragments with respect to different perspectives.

Indeed, slicing can be used in many aspects of program comprehension: Harman, Sivagurunathan and Daninic [HSD98, SHS02] used program slicing in understanding dynamic memory access properties. Komondoor and Horwitz [KH01] presented an approach that use PDG and slicing to find duplicate code fragments in C programs. Henrard et al [HEH⁺98] made use of program slicing in database understanding. Korel and Rilling used dynamic slicing to help understand the program execution [KR97].

4.4.5 Testing

Software maintainers are also faced with the task of regression testing: retesting software after a modification. This process may involve running the modified program on a large number of test cases, even after the smallest of changes. Although the effort required to make a small change may

be minimal, the effort required to retest a program after such a change may be substantial. Several algorithms based on program slicing have been proposed to reduce the cost of regression testing.

A program satisfies a conventional *data flow testing* criterion if all def-use pairs occur in a successful test.

Duesterwald, Gupta and Soffa [DGS92] propose a more rigorous testing criterion, based on program slicing: each def-use pair must be exercised in a successful test; moreover it must be *output influencing*, i.e., have an influence on at least one output value. A def-use pair is output-influencing if it occurs in an *output slice* (a slice with respect to an output statement). It is up to user, or an automatic test generator to construct enough tests such that all def-use pairs are tested.

Kamkar, Shahmerhi and Fritzson [KFS93a] extended the previous work to multi-procedure programs. To this end, they define appropriate notions of interprocedural def-use pairs. The interprocedural dynamic slicing method of [KFS93b, KSF92] is used to determine which interprocedural def-use pairs have an effect on a correct output value, for a given test. The summary-graph presentation is slightly modified by annotating vertices and edges with def-use information. This way, the set of def-use pairs exercised by a slice can be determined efficiently.

In [GHS92], Gupta, Harrold and Soffa describe an approach to regression testing where slicing techniques are used. Backward and forward slices serve to determine the program parts affected by the change, and only tests which execute affected def-use pairs need to be executed again. Conceptually, slices are computed by backward and forward traversals of the CFG of a program, starting at the point of modification.

In [BH93b], Bates and Horwitz used a variation of the PDG notion of [HPR89b] for incremental program testing. Bates and Horwitz presented test selection algorithms for all the vertices and flow-edges test data adequacy criterion. They proved that statements in the same class are exercised by the same tests. This work only considers single procedure programs.

Binkley [Bin97] presented two complementary algorithms for reducing the cost of regression testing that operate on programs with procedures and procedure calls.

4.4.6 Measurement

Cohesion and **coupling** are two important metrics in software measurement.

Cohesion is an attribute of a software unit that measures the “relatedness” of the unit. It has been qualitatively characterized as coincidental, logical, procedural, communicational, sequential and functional; coincidental is the weakest and functional is the strongest.

Several approaches using program slicing to measure cohesion have been presented.

It was Longworth [Lon85] the first to study the use of program slicing as indicator of cohesion.

Ott and Thuss [OT89] then noted the visual relationship that exists between the slices of a module and its cohesion as depicted in a slice profile. Certain inconsistencies noted by Longworth were eliminated through the use of metric slices [OB92, Ott92, OT93, Thu88]. A metric slices takes into account both uses and used by data relationships; that is, they are the union of Horwitz et al's backward and forward slices.

Bieman and Ott [BO93] examined the functional cohesion of procedures using a data slice abstraction. A data slice is a backward and forward static slice that uses data tokens rather than statements as the unit of decomposition. Their approach identifies the data tokens that lie on more than one slice as the "glue" that bind separate components together. Cohesion is measured in terms of the relative number of glue tokens, tokens that lie on more than one data slice, and super-glue tokens, tokens that lie on all data slices in a procedure, and the adhesiveness of the tokens.

Coupling is the measure of how one module depends upon or affects the behavior of another. Harman et al [HOSD] proposed a method of using program slicing to measure coupling. It is claimed that this method produce more precise measurement than information flow based metrics.

4.5 Tools using Program Slicing

In this section we present some tools that uses program slicing to aid at program understanding and comprehension.

4.5.1 CodeSurfer

As referred in subsection 2.5.3, CodeSurfer [Gra08b] is a static analysis tool designed to support advanced program understanding based on the dependency-graph representation of a program. CodeSurfer is thus named because it allows surfing of programs akin to surfing the web [AT01].

CodeSurfer builds an intermediate representation called the system dependency graph (SDG— see section 2.1). The program slices are computed using graph reachability over the SDG. CodeSurfer's output goes through two preprocessing steps before slicing begins [BGH07].

The first identifies intraprocedural strongly connected components (SCCs) and replaces them with a single representative vertex. The key observation here is that any slice that includes a vertex from an SCC will include all the vertices from that SCC; thus, there is a great potential for saving effort by avoiding redundant work [BH03]. Once discovered, SCC formation is done

by moving all edges of represented vertices to the representative. The edgeless vertices are retained to maintain the mapping back to the source. While slicing, the slicer need never encounter them.

The second preprocessing step reorders the vertices of each procedure into topological order. This is possible because cycles have been removed by the SCC formation. Topological sorting improves memory performance — in particular, cache performance [BH03]. After preprocessing, two kinds of slices are computed: backward and forward interprocedural slicing.

Operations that highlight forward and backward slices show the impact of a given statement on the rest of the program (forward slicing), and the impact of the rest of a program on a given statement (backward slicing). Operations that highlight paths between nodes in the dependency graph (chops) show ways in which the program points are interdependent (or independent).

4.5.2 JSlice

JSlice [WR08, WRG] was the first dynamic slicing tool for Java programs. This slicer proceeds by traversing a compact representation of a bytecode trace and constructs the slice as a set of bytecodes; this slice is then transformed to the source code level with the help of Java class files. This slicing method is complicated by Java's stack-based architecture which require to simulate a stack during trace traversal.

Since the execution traces are often huge, the authors of the tool develop a space efficient representation of the bytecode stream for a Java program execution. This compressed trace is constructed on-the-fly during program execution. The dynamic slicer performs backward traversal of this compressed trace directly to retrieve data/control dependencies, that is, slicing does not involve costly trace decompression.

4.5.3 Unravel

Unravel [LWG⁺95] is a static program slicer developed at the National Institute of Standards and Technology as part of a research project. It slices ANSI-C programs. The limitations of Unravel are in the treatment of unions, forks, and pointers to functions. The tool is divided into three main components:

- a source code analysis component to collect information necessary for the computation of program slices;
- a link component to connect information from separate source files together;
- and an interactive slicing component: to extract program components that the software quality assurance auditor can use; and to extract

program statements for answering questions about the software being audited.

By combining program slices with logical set operations, *Unravel* can identify code that is executed in more than one computation.

4.5.4 HaSlicer

HaSlicer [Rod06, RB06] is a prototype of a slicer for functional programs written in *Haskell*. The tool was built for the identification of possible coherent components from monolithic code. It covers both backward and forward slicing using a *Functional Dependency Graph* (FDG), an extension to functional languages of PDG.

In [RB06] the authors discuss how the tool can be used to component identification through the extraction process from source code and the incorporation of a visual interface over the generated FDG to support user interaction.

4.5.5 Other Tools

There are other tools that use program slicing: CodeGenie [LBO⁺07, Lop08] is a tool that implements a test-driven approach to search for code available on large-scale code repositories in order to reuse the fragments found; and GDB-Slice⁶ which implements a novel efficient algorithm to compute slices in GDB (GNU Project debugger) through the GCC (GNU C Compiler Collection) [GABF99].

⁶<http://www.sed.hu/gdbslice/>

Chapter 5

State-of-the-Art: Visualization

*...thought is impossible without
an image.*

Aristotle, 384-322 BC

Nowadays, software systems are growing in terms of size and complexity and their development and maintenance usually involves team work (many people contributing for the same system, but in different parts). This makes the task of programming, understanding and modifying software more difficult, especially when dealing with code written by others. Knowledge of code decays as the software ages and the original programmers and design team move on to new assignments. The design documents are usually out of date because they have not been maintained, leaving the code as the only guide to system behavior. Thus, it is a very time consuming and tedious task to understand complex system behavior from code. Therefore, tools for supporting and making easier these tasks have become essential.

One key aspect to help engineers to deal with complexity and to increase programmers productivity is through Software Visualization (SV). Along the last two decades, several techniques and tools became available to support tasks such as analysis, specification/modeling, coding, testing, debugging, maintenance and understanding. These activities can be assisted through various aspects of software visualization, including visual modeling, visual database query, visual programming, algorithm animation, program visualization, data visualization, and document visualization.

The reason why a chapter on visualization is included in this document is due to the need of finding a powerful visual way to present the semantic/conditional slices produced. As it will be shown along the chapter, displaying the slices of a program in a visual way, instead of textual, is not a novelty. However, due to the complexity of the slicing algorithms that

will be presented in the next chapters, and because usually in Program Verification visualization techniques are not used, it was considered of great aid to include them in the PhD work here reported. In this context, it was taken profit of the visualization techniques for both components: slicing and verification processes; also animation features were employed.

Structure of the chapter. In Section 5.1, the area of software visualization is characterized: some definitions of SV are presented, and the process of extract information to visualize is also discussed. Because along the years many taxonomies came up in an attempt to classify this area, in Section 5.2, a brief overview of the most important taxonomies is presented. In Section 5.3, are discussed the main techniques employed along the years to display information. Section 5.4 presents some of the application fields and tools of SV.

5.1 Characterizing Software Visualization

Software visualization use graphical techniques to make software visible through the display of programs, program artifacts (use cases, class diagrams, and so on), and program behavior. The essential idea is that the visual representations can make the process of understanding software easier. Pictures of the software can help project members to remember the code and new members to discover how the code works.

In [Zha03], *Kang Zhang* discusses how each phase of software engineering process can be influenced by SV techniques. Figure 5.1 depicts the relationship between each phase and some visualization artifacts.

During the first phase, software managers use different data visualization resources, such as Gantt charts, to outline the project scheduling and the different milestones.

In the second phase, for the requirement analysis and specifications, the use of state charts and class diagrams is common.

In the third phase, when the overall software architecture is established through system and software design, visual modeling techniques can play an important role by using various types of architectural diagrams, such as class diagrams and collaborative diagrams. Visual languages as well as animation techniques can also be particular useful to the explanation of an algorithm's behavior.

In the fourth and fifth phases, the software domain can be coded via visual programming. Both unit testing and integrating testing may be done through the use of program slicing techniques and be visualized through program visualization artifacts based on graph formalisms, such as call graphs and dependence graphs.

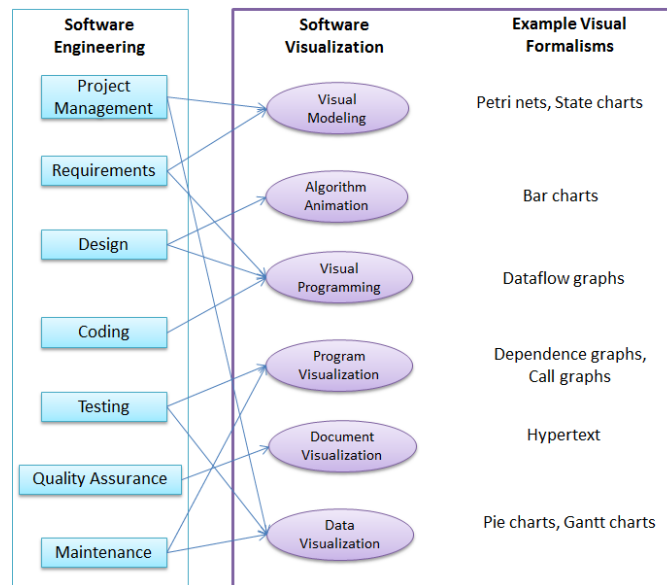


Figure 5.1: Software Engineering Process and Software Visualization Role

In the next phase, software documentation can give a contribute for quality assurance of any software product. A graph showing the software's documentation as an intuitive road map, could be useful for tutorial, guiding or diagnostic purposes.

In the final phase, the longest in software life cycle, visualization techniques can be used to reveal bugs or requirement errors. Program understanding and analysis could be achieved more effectively through graphical representations. Also, performance evaluation and comparison can be conducted effectively through data visualization (also called statistical visualization). Program visualization differs from data visualization in the sense that the in the former visualizations correspond directly to the program semantics (e.g. nodes in a call graph represent procedure/functions and edges represent call relationships) and in the latter correspond to program measurements (a segment in a pie chart is significant only in its size and what it measures).

Along the years, the term *software visualization* has many meanings depending on the author. Below there are some definitions for software visualization than can be found in literature.

“ Software visualization is the use of crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both human understanding and effective use of computer software.

JOHN STASKO ET AL

Software visualization: programming as a multimedia experience [SDBP98]

“ Software visualization refers to the use of various visual means in addition to text in software development. The forms of development means include graphics, sound, color, gesture, animation, etc.

KANG ZHANG

Software Visualization: From Theory to Practice [Zha03]

“ Software visualization encompasses the development and evaluation of methods for graphically representing different aspects of software, including its structure, its abstract and concrete execution, and its evolution.

SOFTVIS CONFERENCE SITE

<http://www.st.uni-trier.de/~diehl/softvis/org/softvis10/>

“ ... Program visualization is defined as a mapping from programs to graphical representations.

GRUIA-CATALIN ROMAN AND KENNETH C. COX

Program Visualization: The Art of Mapping Programs to Pictures [RC92]

“ ... we define software visualization as the mappings from software to graphical representations.

RAINER KOSCHKE

Software visualization for Reverse Engineering [Kos02]

For the purposes of this document, software visualization can be taken to mean any form of program visualization that is used after the software has been written as an aid to understanding. It will be considered the following definition of SV:

“ Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.

CLAIRE KNIGHT AND MALCOLM MUNRO

Comprehension with[in] Virtual Environment Visualisations [KM99]

Thus, in addition to program (source code), artifacts like requirements and design documentation, changes to source code and bug reports could be used. Along the years different methods and uses of computer graphical representations were investigated in order to explore the various aspects of software (e.g. its static structure, its concrete and abstract execution, and its evolution). According to [Die07], SV researchers are concerned with:

- *Visualize the structure*: structure means the static parts and relations of the systems (those that can be computed without running the program). These parts includes the program code and its data structures, the static call graph, and the organization of the program into modules.
- *Visualize the behavior*: behaviors refers to the execution of the program with real and abstract data and comprises the visualization of control and data flow.
- *Visualize the evolution of the software*: evolution refers to the process where the program code changes to extend the functionality of the system or to remove bugs.

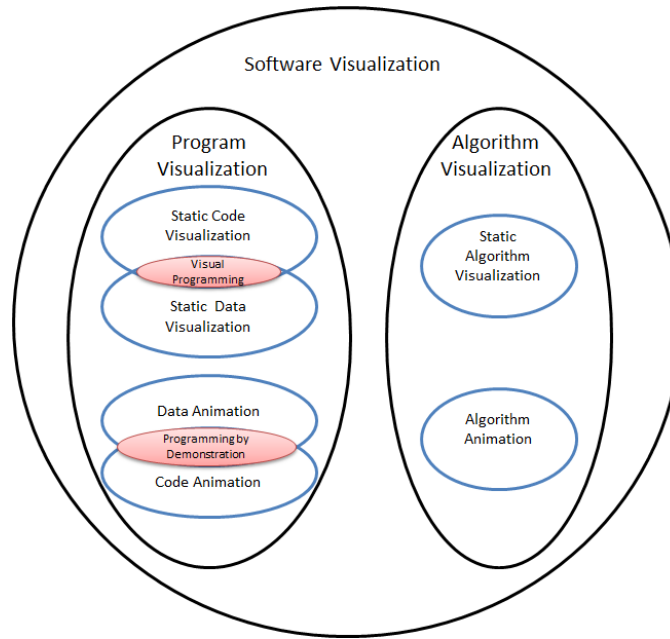


Figure 5.2: Venn diagram showing the relationships between the various forms of software visualization [BBI98]

This is directly related with the discussion taken by *Ball and Eick* in [BE96], when they refer that there are three basic properties of software to be visualized: *software structure*, *runtime behavior* and *the code itself*.

As previously referred, SV can be defined as the discipline of generating visual representations for various aspects of software as well as its development process. Figure 5.2 shows the relationships between the various forms of software visualization, according to Price’s perspective [BBI98].

According to *Price et al*, *program visualization* is the *visualization of actual program code or data structures in either static or dynamic forms*. Once again, there is no consensus about this definition. Consider the following examples to illustrate the left part of the Venn diagram (program visualization):

- *Static code visualization*: it can include some kind of graphical layout illustrating call graphs or control flow graphs.
- *Static data visualization*: it can display the content of data structures (like linked lists) as a “boxes and arrows” diagram.
- *Data animation*: it might use the same diagram as in previous item (data visualization= but with the content of boxes and the arrows changing along the program execution).

- *Code animation*: it might simply highlight lines of code in the IDE as they are being executed.

The authors consider that both *visual programming* and *programming by demonstration/example* are not subsets of program visualization but they have a partial overlap with it. Visual programming can be described as the kind of programming that seeks to make programs easier to specify by using a graphical notation for the language constructors and their interconnection. Programming by demonstrations or by example is related with visual programming and it is defined as the specification of a program with user demonstrated examples.

The goal of algorithm visualization is to show abstractly how a program operates through the “visualization of higher level abstractions which describe software” [BBI98]. This category is also divided into static and dynamic visualizations:

- *Static algorithm visualization*: it often consists of a snapshot, a trace of one execution of the algorithm.
- *Algorithm animation*: it shows how an algorithm works by depicting its fundamental operations, graphically or resorting to the ear (usually sound). It might show the swap of two items as a smooth animation.

But the creation of images in software visualization is only the last step in the *visualization pipeline* (Figure 5.3). To display graphical objects associated to some information, there is the need to collect the information intended to show and analyze it in order to filter according to a specific goal. The information produced by one stage is used as input by the next stage:

Data acquisition: different information sources related to the software is usually available: besides the source code, there is the documentation, the software model, test results and mailing lists. Various methods to extract and gather the relevant data from these sources are used (in Chapter 2 some of these techniques were discussed).

Analysis: after gather the extracted data, and because usually it is too much information to display without any kind of treatment, there is the need to perform different kinds of analysis and filtering (slicing — see Chapter 4 for more details about this subject — or statistical methods are reduction operations that can be done).

Visualization: once the information is gathered and filtered, it can be mapped onto a visual model, i.e., transformed into graphical information, and then rendered onto the screen. In interactive visualizations, the user

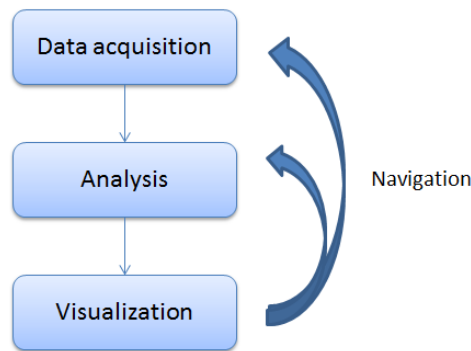


Figure 5.3: Visualization pipeline

can control the previous steps of the pipeline on the basis of the graphical output produced earlier.

According to *Young and Munro* [YM98] there is a list of desirable properties in a software visualization system to make it “good”. These properties are:

- *Simple navigation with minimum disorientation*: the visualization should be structured and should include features to lead the user in navigating the visualization.
- *High information content*: “Visualizations should present as much information as possible without overwhelming the user”.
- *Low visualization complexity, well structured*: well structured information should result in easier navigation. Low complexity trades-off with high information content.
- *Varying levels of detail*: granularity, abstraction, information content and type of information should vary according to the users interests.
- *Good use of visual metaphors*: metaphors are a way to provide cues to understanding.
- *Approachable user interface*: the user interface should be flexible and intuitive, and should avoid unnecessary overheads.
- *Integration with other information sources*: it is desirable to be able to link between the visualization and the original information it represents (the source code).
- *Good use of interactions*: interaction provides mechanisms for gaining more information and maintaining information.

- *Suitability of information*: “a good level of automation is required in order to make the visualizations of any practical worth”.

Albeit its intrinsic importance, SV topic is so broad and complex that several researchers have proposed taxonomies to classify software visualization research and tools. Next section is devoted to a brief overview of such taxonomies.

5.2 Taxonomies

On one hand, SV taxonomies allow to compare systems; on the other hand, SV taxonomies are useful to understand all the issues that must be taken into account.

Along the years, many researchers come up with valid taxonomies to help classify, quantify and describe different types of software visualization. Each taxonomy emphasizes different SV aspects, for instance specific software characteristics, human activities, and so on.

In this section, an overview of the most important SV taxonomies are presented.

Brown [Bro88] introduced a visualization approach focused on animations. He describes his proposal using three main axes: *Content*, *Transformation* and *Persistence*.

The first is concerned with the way followed for representing the program. This characteristic is divided in *Direct* (in which the source code is directly represented using graphical artifacts) and *Synthetic* (in this case a source code abstraction is graphically depicted).

The second refers to the animation process used in the visualization. It can be *Discrete* (a series of snapshots are shown) or *Incremental* (a smooth technique is employed to produce the transition between snapshots).

Finally, *Persistence* is related with the possibility of holding the process history.

Myers [Mye90] introduced a taxonomy for program visualization which identifies six regions arranged in a 2×3 matrix as depicted in Figure 5.4. His simple taxonomy has two main axes:

- *The kind of object that the visualization system attempts to illustrate.*
- *The type of information shown.*

The first component consists of data, code and algorithms, and the second one is concerned with the static and dynamic information. The combination of these axes produces the following visualization sorts:

- **Data-static**: it follows the same approach that for static code visualizations. The main idea consists in presenting in a suitable way the system data objects and their values.

	Static	Dynamic
Data		
Code		
Algorithm		

Figure 5.4: Myers taxonomy

- Data-dynamic: similar code-dynamic visualizations, it consists of animations aimed at showing the variables and their values at runtime. This topic is concerned with the strategies for depicting how data changes through the time.
- Code-static: visualizations such as flowcharts.
- Code-dynamic: software animations and strategies to show the code segment used in one specific execution.
- Algorithm-static: generation of snapshots of algorithms.
- Algorithm-dynamic: algorithm animations for presenting an integral dynamic view of each component used at runtime.

Price et al introduced a more comprehensive taxonomy in [PBS93]. Their taxonomy is based upon a tree structure where each leaf provides a different and orthogonal classification criterion. Figure 5.5 depicts the two level taxonomy (to see the details about this second level and subagent levels, please read [PBS93]).

The first level of this taxonomy contains six categories:

- Scope: *What is the range of programs that the software visualization system may take as input for visualization?* Scope relates to the source program and the specific intent of the software visualizer.
- Content: *What subset of information about the software is visualized by the software visualization system?* Content describes the particular aspects of the software that is visualized.
- Form: *What are the characteristics of the output of the system (the visualization)?* Form is concerned with the elements used in the visualization (e.g. graphical elements, colors, views, etc.).
- Method: *How the implementation is specified and how the system works?* Method is concerned with the strategies used to specify the visualization (e.g. fixed, customizable, etc).

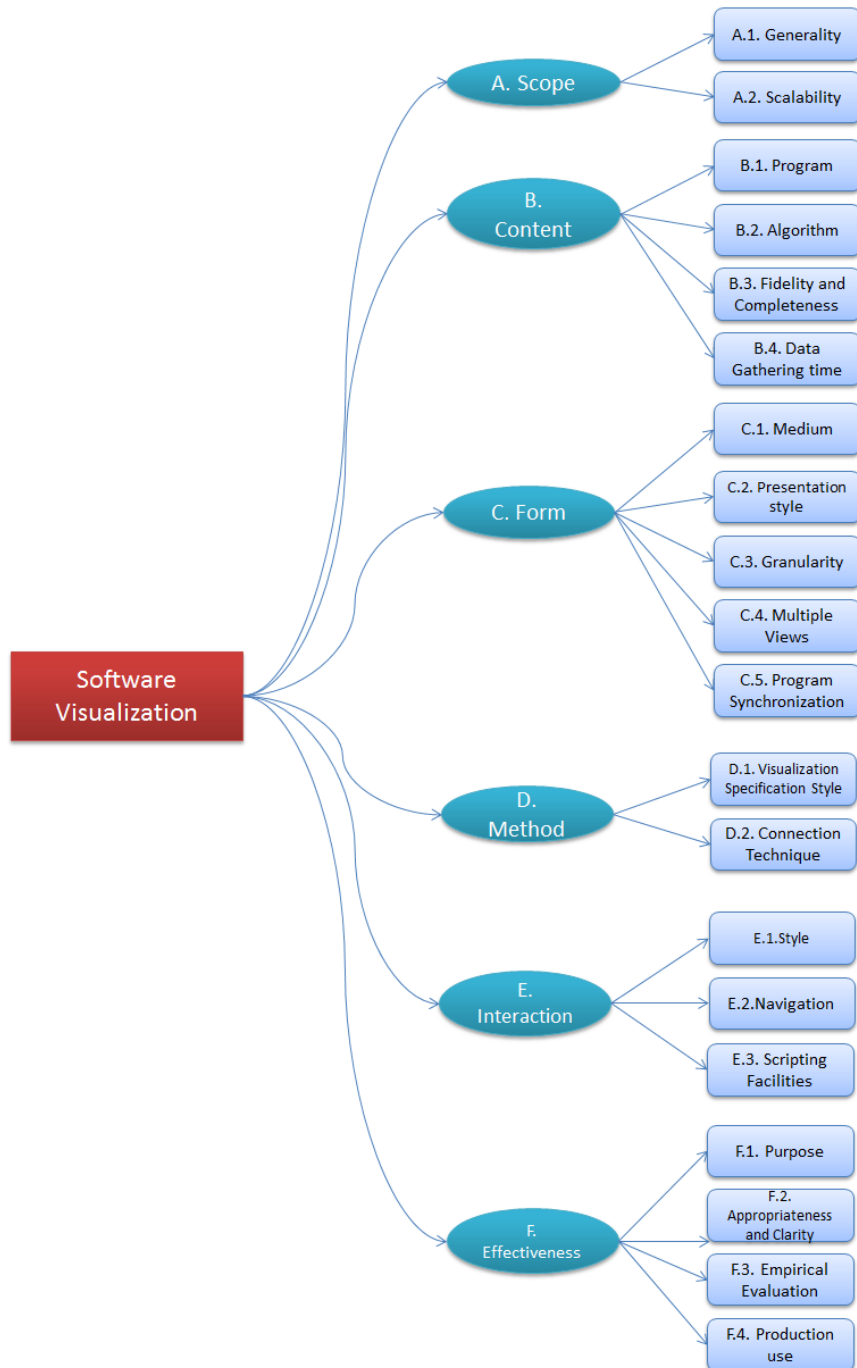
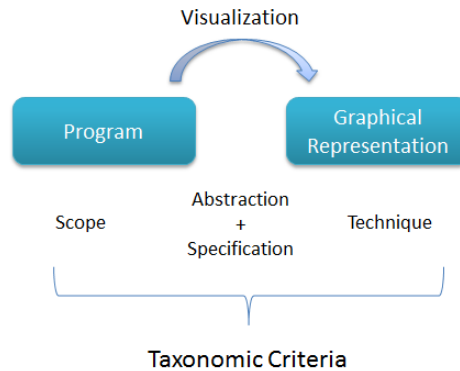


Figure 5.5: Price et al taxonomy

Figure 5.6: *Roman and Cox* taxonomy

- Interaction: *How does the user of the system interact with it and control it?* Interaction is concerned with the techniques used to control the visualization (e.g. navigation, temporal control mapping, etc).
- Effectiveness: *How well does the system convey to the user?* Effectiveness gives the characteristics to assess the visualization quality.

Another taxonomy is the one defined by *Roman and Cox* [RC93], based on their earlier work [RC92], and it uses four criteria based in the SV model depicted in Figure 5.6:

- Scope: *What aspect of the program is visualized?* Scope describes the program aspects to visualize.
- Abstraction: *What kind of information is conveyed by the visualization?* Abstraction describes the visualization specification degree.
- Specification Method: *How the visualization is constructed?* Specification method explains which are the mechanisms used by the animator for building the visualization.
- Technique: *How is the graphical representation used to convey the information?* Technique is concerned with the effectiveness of the information presentation.

Later on, in 2002, *Maletic et al* in [MMC02], proposed a realignment to the existing taxonomies in order to reflect the new challenges and issues of the recent software, in particular, software engineer tasks of large-scale development and maintenance. They proposed a five dimensioned taxonomy to classify software visualization systems and these dimensions reflect the *why, who, what, where* and, *how* of the software visualization. The authors

Dimension	<i>Roman and Cox</i> [RC92]	<i>Price et al</i> [PBS93]
Task		F.1. Purpose
Audience		F.1. Purpose
Target	Scope Abstraction	A. Scope B. Content
Representation	Specification Method Interface Presentation	C. Form D. Method E. Interaction F. Effectiveness
Medium		Form

Table 5.1: Overview of the relations between the proposed dimensions and the criteria defined in the taxonomies of *Roman and Cox* and *Price et al* respectively [MMC02].

argue that this taxonomy accommodates a larger spectrum of software visualization systems than the other taxonomies defined so far (for example, algorithm animation and visual programming tools).

They pose the five dimensions through the following questions:

- Tasks: *why is the visualization needed?*
- Audience: *Who will use the visualization?*
- Target: *what is the data source to represent?*
- Representation: *how to represent it?*
- Medium: *where to represent the visualization?*

The authors also discuss how these dimensions could be mapped into the ones presented by *Roman and Cox* and by *Price et al*. Table 5.1 displays this mapping. At the end, the authors analyze a set of software visualization systems and categorize them under this taxonomy.

In 2008, *Beron et al* proposed a new taxonomy towards Program Comprehension [BdCP⁺08]. According to the authors, the existing taxonomies do not cover the kind of visualization required by Software Visualization systems concerned with Program Comprehension. In particular, these taxonomies do not incorporate dimensions to assess the visualizations oriented for both Problem and Program Domains.

The authors proposed an extended taxonomy based on six main dimensions:

- *Scope*: concerned with the Problem Domain characterization. It is divided into the following categories: stimulus/response, concepts/relations, subsystem relations, and behavior.

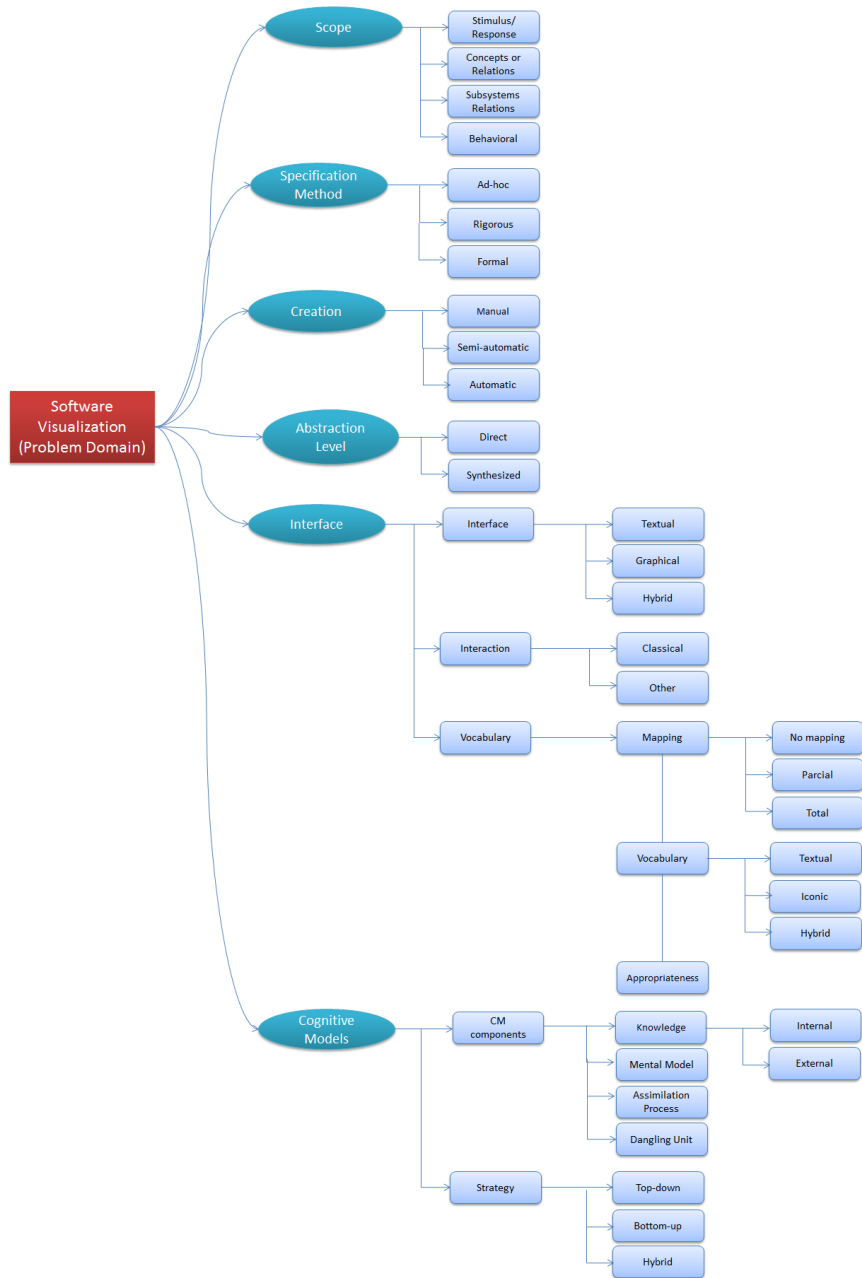
- *Specification Method*: concerned with how to link the Problem Domain with the Program Domain components. It is divided into the following approaches: ad-hoc, rigorous and formal.
- *Kind of Creation*: concerned with the strategy used to create the Problem Domain visualization. It is divided into manual, semi-automatic and automatic.
- *Abstraction Level*: concerned with the level of detail devoted to show the Problem Domain characteristics. It is divided into direct, structural and synthesized.
- *Interface*: concerned with artifacts used to display the Problem Domain visualizations. It is divided into kind of interface (textual, graphical or hybrid), type of interaction (classical or innovative) and vocabulary (textual, iconic or hybrid).
- *Cognitive Models*: concerned with the cognitive factors. It is divided into Cognitive Model components (internal and external knowledge, mental model, assimilation process and dangling unit), and learning strategies (top-down, bottom-up and hybrid).

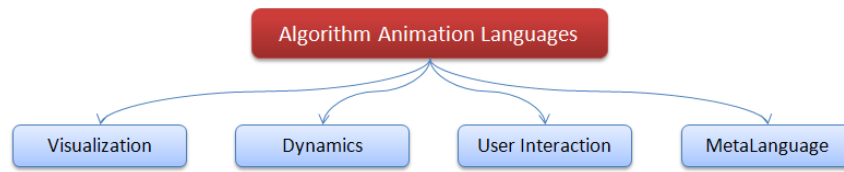
Although some of these dimensions are similar to those defined previously for Program Domain, their means are significantly different. Figure 5.7 depicts the complete taxonomy.

More recently, in 2010, *Karavirta et al* [KKMN10, KKM06] extended the Price's taxonomy to cope with algorithm animation languages (a formal language description that specifies an algorithm animation). Their extended taxonomy is based on four main dimensions (see Figure 5.8):

- *Visualization*: concerned with the variety of supported object types (the building blocks used in the animation), and the ways to position and style the objects.
- *Dynamics*: concerned with level and versatility of the language features that can be used to modify the visualization and thus create the animation.
- *User Interaction*: concerned with the level and type of interaction provided to the end user.
- *MetaLanguage*: concerned with the properties of the language not directly related to algorithm animation but still helpful in the process of creating useful animation content.

The authors consider that this taxonomy is useful for comparing different languages with each other and understanding their strengths and limitations.

Figure 5.7: *Beron et al* taxonomy

Figure 5.8: *Karavirta et al* taxonomy (parcial)

5.3 Techniques

Along the years, the techniques used to display information evolved significantly.

In the beginning of computers era, computer programs were conventionally represented as sequences of 0s and 1s. Despite the fact that modern computers today still use this form, nowadays computer programs are written using high level languages — the binary language is now represented using English keywords. Also, currently modern programming environments use other elements such as collapsible items, pictures, and animated tracking (e.g. debugging).

Furthermore, considerable efforts were done in the field of software visualization in order to display information in a natural and intuitive way, to help the user to better understand and get insights about the data being visualized.

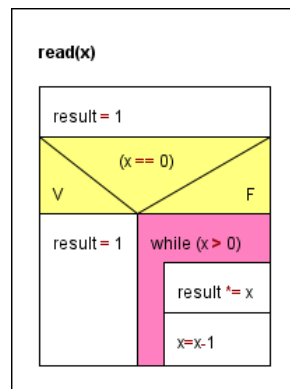
Because textual representations are a poor and limited way to visualize a program (imagine the case of large programs composed by millions of lines of code), researchers continually strived to find better visualization techniques. Computer programs as graph representations became a popular way to show different perspectives of a program.

Essentially, visualization techniques can be divided in: *textual* and *graphical*. In both of these techniques, *static* or *dynamic* techniques could be used to get the information needed to display.

Some approaches to software visualization were reviewed and hereafter presented (with the purpose of choose the technology to use in the implementation of GamaSlicer).

5.3.1 Nassi and Shneiderman Diagram

Was in the 70s that the search for graphical techniques started to appear with the work of Nassi and Shneiderman [NS73]. They proposed a diagram in a rectangular form subdivided into smaller rectangles that represent instructions. A conditional instructions is separated by two triangles representing the alternatives (yes/no). The flow of program is representing by going down the rectangle. These diagrams are also called *structograms*, as they show the program structure. Figure 5.9 depicts an example of this

Figure 5.9: *Nassi-Shneiderman* diagram for Factorial program

flowcharts, representing the computation of a factorial number. A conditional and a loop are represented. This picture was built using a tool called *Structurizer* and is available at <http://structurizer.fisch.lu/>.

Although very naive, this so simple approach to the visual representation of the control flow of a program, two motives make it worth of mentioning. It was the first attempt with very poor graphical resources. Nowadays this kind of visualization was recovered by *Scratch*, [MSABA10] one of the most recent Visual Programming Languages.

5.3.2 Information Murals

An information mural is a graphical representation of a large information space that fits entirely within a view. This technique is generally useful for visualizing trends and patterns in the overall distribution of information. In this technique, two main categories can be found: line and pixel representations; and execution mural.

Line and Pixel representations This technique was developed by *Ball and Eick* to visualize program text, text properties and relationships [BE96]. Each line of the program text is reduced to a single row of pixels, where length and indentation corresponds to the original code.

They generalize this technique to increase the information density by a factor of ten. Each column represents a single file. Each line of code uses color-coded pixels ordered from left to right in rows within the columns. Figure 5.10 depicts an example of line (top) and pixel (bottom) representation.

Execution Murals Execution murals is a technique introduced by *Jerdling and Stasko* that provides a quick insight of the various phases of the

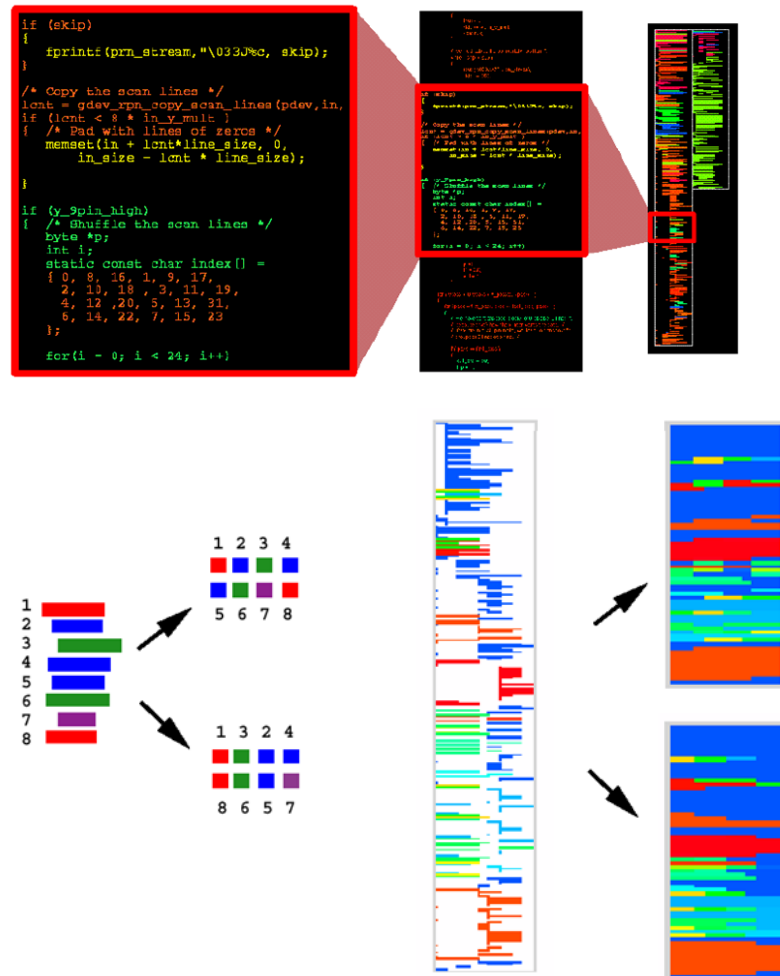


Figure 5.10: Line and pixel representation of a program using *Ball and Eick* technique

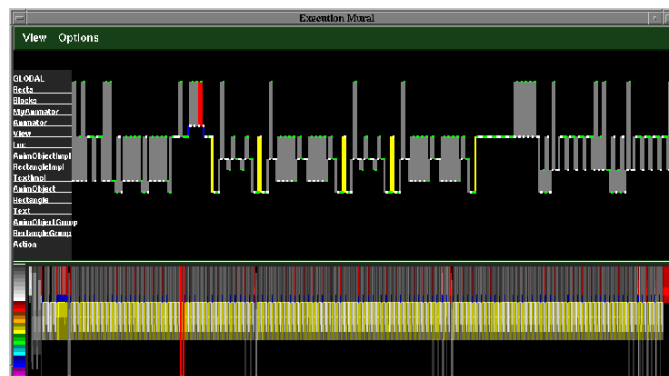


Figure 5.11: Execution mural of message traces for an object-oriented program

execution of object-oriented programs [JS98]. Figure 5.11 shows an execution mural of message traces. The upper portion of the view is the focus area where a sub-set of the messages can be examined in detail. The bottom portion of the view is a navigational area that includes a mural of the entire message trace, and a navigation rectangle indicating where the focus area fits in the entire execution.

5.3.3 Graphs

Graphs are one of the most common representations in software visualization. Usually, nodes represent block and instructions and edges indicate the flow. Different types of graphs were proposed along the years in order to show different perspectives of a program, depending of the goal of the analysis being done:

- Abstract Syntax Tree (see Definition 1)
- Control Flow Graph (see Definition 2)
- System Dependence Graph (see Definition 9)
- Call Graph (see Definition 10)
- Value Dependence Graph (see Definition 11)
- Module Dependence Graph (see Definition 12)
- Trace Flow Graph (see Definition 14)

All these kind of graphs face a problem with respect to visualization: where to place each node in order to obtain the maximum readability and effectiveness. Several aesthetic criteria concerning *graph layout* have been investigated in this context [PCJ96]:

- *Crossing minimization*: a graph should be drawn with as few crossings as possible.
- *Bends minimization*: edges should have as few bends as possible.
- *Area minimization*: the area of the drawing should be small and there should be a homogeneous density or an even distribution of the nodes.
- *Length minimization*: shorter edges are more easy to follow. It should be minimized the total edge length and the length of the longest edge.
- *Angle maximization*: small angles between outgoing edges and in bends make it difficult to discriminate the edges.
- *Symmetries*: symmetries in the underlying graph should be reflected in the diagram.
- *Clustering*: for large graphs, parts of the graph which are strongly interconnected — called clusters — should be drawn separate from other parts. Edges drawn in a cluster should be shorter than those connecting different clusters.

Graph drawing techniques differ in the kinds of graphs they are suitable for and in the basic principles underlying the drawing algorithm.

Following are briefly described some of the most important graph layout algorithms used in the most popular software visualization systems. The images shown to illustrate each one of the layouts are either produced by yFiles¹ or by Nevron² tools.

- *Orthogonal layout*: the edges run either horizontally or vertically and edge inclination must have an angle of 90 degrees. The goal with this kind of layout is to minimize the number of edges crossings and deviations (see Figure 5.12).
- *Force-directed Layout* (also called *Force-based Layout*): the nodes are placed in a way so that all the edges are of more or less equal length and there are as few crossing edges as possible. They are usually selected for undirected graphs and are specially target for simulating physical and chemical models. The force-directed algorithms assign forces amongst the set of edges and the set of nodes. The most common is to assign forces as if the edges were springs and the nodes are electrically charged particles. The entire graph is then simulated as if it was a physical system. The electrical force repels the vertices which are close to each other. This is repeated iteratively until the

¹Available at <http://www.yworks.com>

²Available at <http://www.nevron.com>.

system comes to an equilibrium state; i.e., their relative positions do not change anymore from one iteration to the next (see Figure 5.13).

Another common forces are *magnetic* (edges are interpreted as magnetic needles that align themselves according to a magnetic field) and *gravitational* (all nodes are attracted to the bary center of all the other nodes).

- *Hierarchical Layout*: the nodes are placed in hierarchically arranged layers such that the (majority of) edges of the graph show the same overall orientation, for example, top-to-bottom. Additionally, the ordering of the nodes within each layer is chosen in such a way that the number of edge crossings is small (see Figure 5.14).
- *Tree Layout*: used for directed graphs that have a unique root element (trees). Starting with the root node the nodes are arranged either from top to bottom, left to right, right to left, or bottom to top. The edges of a graph can either be routed as straight lines or in an orthogonal fashion (see Figure 5.15).
- *Radial Layout*: organizes the graph in concentric circles of nodes. The vertices are placed in the center and their descendants are placed on the next circle and so on. It produces a straight line graph drawing (see Figure 5.16).

Other graph layouts (as well as variations of the ones presented) can be used depending of the purpose of the analysis: *hyperbolic layout*, *symmetric layout*, *circular layout* and so on.

But the problem of displaying a graph becomes more complex when a visualization system evolves from static visualizations to dynamic ones (*graph animation* systems).

Usually, during an animation process using graphs, it is necessary to add and delete edges and nodes. The naïf approach, to display a sequence of graphs, is to re-calculate the whole layout in each update. This results in an additional aesthetic criterion known as *preserving the mental map*. Mental map refers to the abstract structural information that a user forms when looking at the layout of a graph [DG02]. Is this mental map that facilitates the navigation in the graph and its comparison with other graphs. So, an obvious challenge is how to display each one of these graphs in order to preserve the basic layout.

In [DGK00], Dieh *et al* proposed a solution: given a sequence of n graphs they compute a global layout suitable for each one of the n graphs. In the simplest case, this global layout can match with the super-graph of all graphs in the sequence. This approach is called *Foresighted Layout*.

A general problem associated with most of the current graph layout techniques is that they do not scale. In general, it is almost impossible to

create a graph with thousands of nodes and keep its consistence or try to minimize edge crossings.

Thus, often the most common and practical solution is to layout a spanning tree for the graph. A spanning tree is a technique that segments a large graph in smaller ones. A list of algorithms to compute spanning trees for graphs can be found in [Jun07].

5.3.4 Visual Metaphors

Other common technique used in SV systems are the *metaphors*. A metaphor can be defined as “a rhetorical figure whose essence is understanding and experiencing one kind of thing in terms of another” [LJ80]. Metaphors might be abstract geometrical shapes or they might be real-world entities. While it is true that a user would be more familiar with a real-world metaphor, the visual complexity of the metaphor should not affect the effectiveness of the visualization. The goal of such metaphors is to evoke mental images to better memorize concepts and to exploit analogies to better understand structures or functions. The goal of software visualization is not to produce neat computer images, but computer images which evoke mental images for comprehending software better. “Finding new metaphors thus will not just produce better visualizations, but it will also improve the way we talk about systems.” [Die07]

When using metaphors they should be:

- *Consistent*: the mapping from software artifacts to the representations should be consistent throughout the visualization. Multiple software artifacts cannot be mapped into the same metaphor. Similarly, a software artifact cannot be mapped to multiple metaphors.
- *Semantically rich*: the metaphors should be rich enough to provide mappings for all aspects of the software that need to be visualized. There should be enough objects or equivalent representations in the metaphor for the software entities that need to be visualized.

For geographic information, maps are widely used to display spatial information. Thus, the *map metaphor* is often applied to display multi-dimensional abstract information by selecting two of the dimensions to span the 2D space as well.

The *landscape metaphor* span the 3D space and represents abstract entities or relations using hills, valleys, rivers and streets. The *city metaphor*, which can be seen as part of the landscape metaphor, represents abstract entities and relations by houses, buildings, towers and streets.

Relational information is often represented by the *galaxy* or *solar system metaphor*. Such visualization can be produced using the force-directed layout algorithm (see subsection 5.3.3).

In the sequel it will be presented a short list of systems that follow this approach and produce different implementations for this metaphoric concept.

CallStax, by *Young and Munro* [YM98] was one of the first attempts to produce visualizations using abstract 3D geometrical shapes. **CallStax** showed the visualization of the calling structure of C code (essentially the same information as a call graph) with colored stacks of blocks showing the routes through the graph.

FileVis [You99], creates a view of the software system where source code files are represented individually as floating platforms around a central point which represents the connectivity of the source code files.

ImsoVision [MLMD01] is a system that creates in a cave a 3D visualization of classes, their properties, their dependencies and aggregations. As the cave is a room where the visualization is projected on the walls, the user can enter the room and interact with the 3D scene. Classes are represented by platforms, methods by columns, and attributes by spheres put on the top of the platforms. The platforms of sub-classes are placed next to their super-classes. Dependencies and aggregations are shown as flat edges. Different properties are represented by colors.

Software Landscapes [BNDL04] uses islands instead of platforms and skyscrapers instead of simple columns.

In **Software World** [KM99], the world represent the software system as a whole, a country represents a package, a city represents a file, a district within a city represents a class, and a building represents a method. The size of the building indicates the number of lines of code, the doors represent its parameters, the windows represent the variables declared in the method, and the color of the building indicates whether the method is private or public.

CodeCity [WL07] also follows the approach of representing a software system as a city.

Vizz3D³ is an open-source tool where houses represent and edges indicate function calls. The size of the house corresponds to the number of lines of code. The platforms represent directories or files containing functions. Icons show additional information. For example, a garbage can indicate dead code, a wheel indicates access to global variables and a lock indicates security issues. Flames used as icons and as textures of houses indicate that the McCabe's cyclomatic complexity exceeds a threshold.

Next section is dedicated to the application fields and tools of software visualization.

³<http://vizz3d.sourceforge.net/>

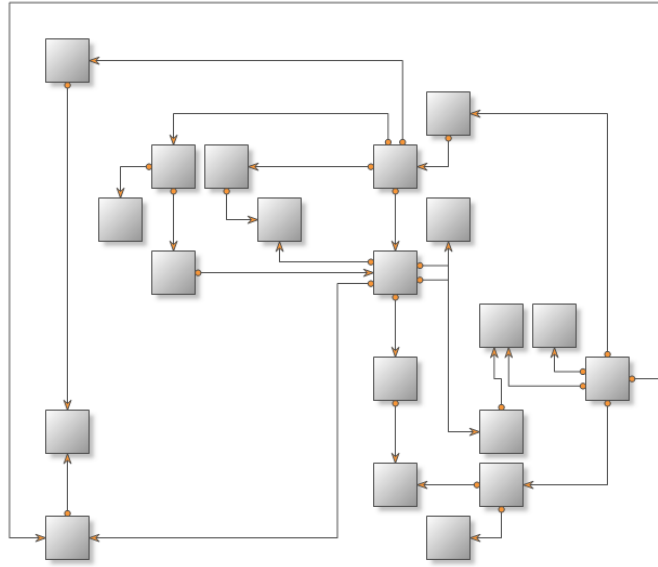


Figure 5.12: Orthogonal Layout produced by Nevron

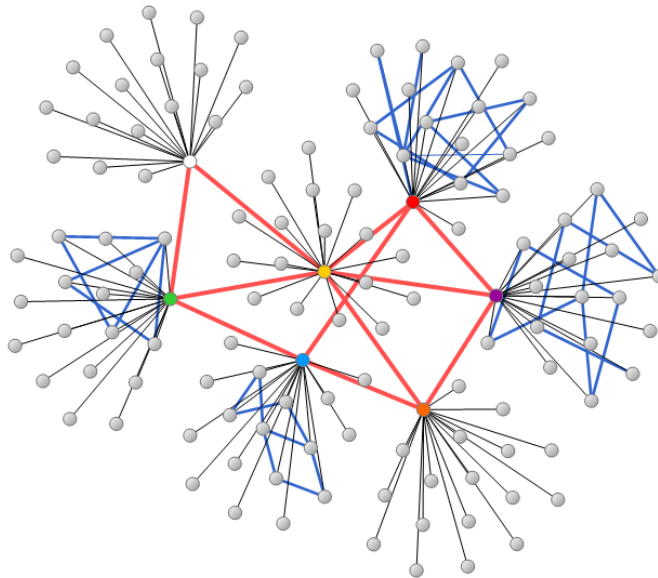


Figure 5.13: Force-Directed Layout produced by Nevron

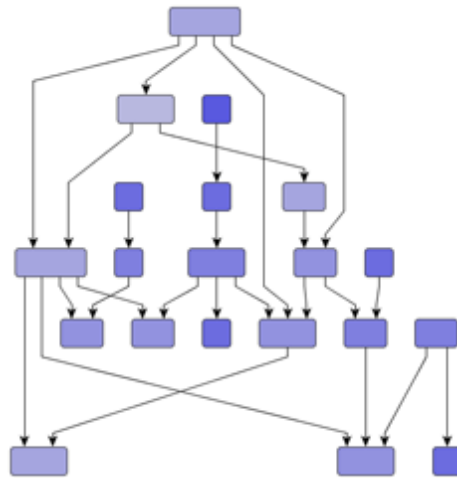


Figure 5.14: Hierarchical Layout produced by yFiles

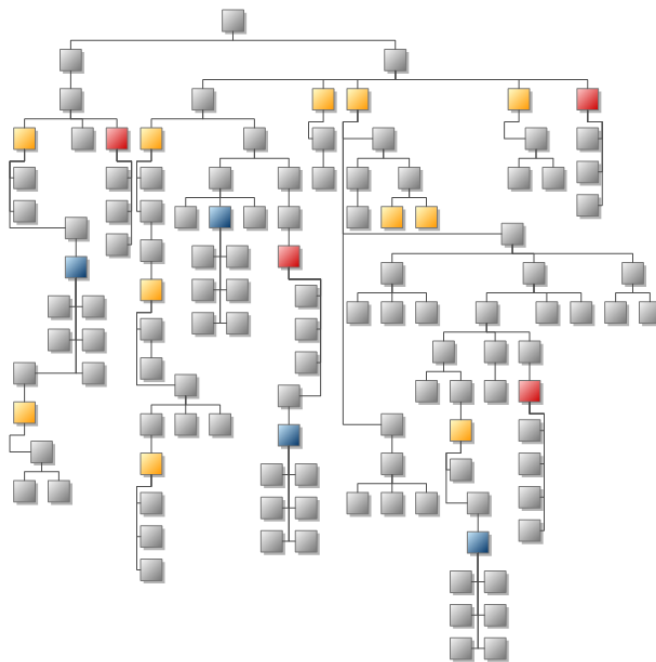


Figure 5.15: Tree Layout produced by Nevron

5.4 Applications and Tools

Based on concepts and developments in information visualization, usability, and software engineering, many systems were developed targeting different goals. In this section, some of the application fields and tools of SV are

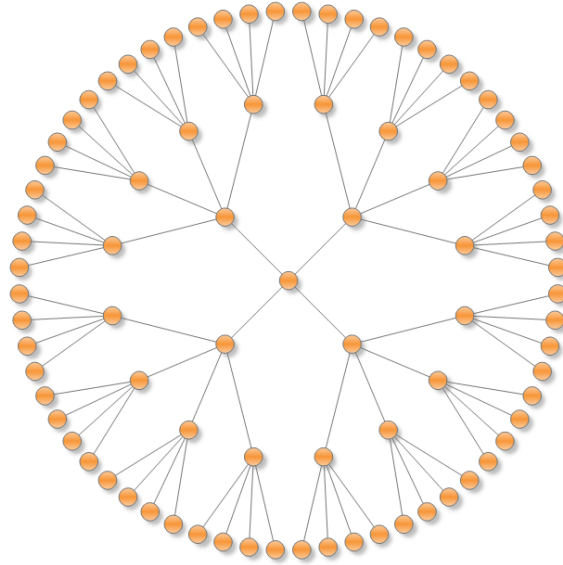


Figure 5.16: Radial Layout produced by Nevron

presented.

Because the range of tools currently available is too extensive, the list of tools hereafter cited do not intend to be exhaustive. The ones referred in this section (in particular, subsections 5.4.3 and 5.4.4) are related with the work reported in this document (tools that allows to visualize slices of programs and animate algorithms).

5.4.1 Program Comprehension

Brooks defined *comprehension as the reconstruction of the domain knowledge used by the initial developer* [Bro77]. The comprehension of software systems both during and after development is a crucial component of the software process.

In particular, the understanding of legacy systems is essential for further development, maintenance and re-engineering of the system. Usually, the system's architecture is poorly documented and so, the only trustworthy source of information is the system implementation. Thus, the architecture has to be retrieved and preferentially in an automated way — program analysis is a way to automatize this task. But, the results of such analysis must be presented in a form that is intuitive to the engineer. Therefore, the program analysis can be combined with software visualization to attain a more effective comprehension of the system.

According to *Oliveira et al* [OPdCB10], an effective program comprehension is reached when it is possible to view and relate what happens when the program is executed, synchronized with its effects in the real world

concepts. This enables the interconnection of program's meaning at both problem and program domains. To sustain this statement the authors proposed a tool, *Alma*², which provides and synchronizes views at both domains. *Alma*² takes a program, written in a **Domain Specific Language** (DSL), and animates it (through an abstract interpretation), displaying in a synchronized mode what happens internally during the execution of each statement and what are the effects of the execution over a graphical representation of the problem domain. Thus, *Alma*² enables to study the behavior of domain specific programs, observing the synchronized visualizations of both domains.

Rilling and Mudur [RM05] proposed a **SV** system, *MetaViz*, that uses program slicing (as discussed in Chapter 4, slicing enables to focus on a specific aspect of a given program) for deriving program-structure attributes and 3D techniques to help visualization-based analysis of source code structure. Through *MetaViz* the authors demonstrate that the use of 3D visuals intuitively comprehensible, actually communicate information about relative component complexity. Thus, they have shown that this technique enhance comprehension of the program structure.

Marcus et al [MFM03] proposed a software visualization system, *sv3D*, based on 3D metaphors to represent large software systems and related analysis data. The tool extends *SeeSoft* pixel representation [ESS99] to a 3D space. Besides this extension, they added new representations coupled with advanced user interaction facilities. Some of these enhancements include:

- Creation of 3D renderings of the raw data.
- Mapping of software artifacts and their attributes to the 3D metaphors, at different abstraction levels.
- Moreover, the system accepts a simple and flexible input in XML format, enabling the output of various analysis tools can be translated to *sv3D* input format.

With *sv3D* the authors brought together results from information visualization, human computer interaction, and program comprehension.

Lanza et al [LDGP05] proposed a system, *CodeCrawler*, that is mainly targeted to visualize object-oriented systems. It provides visualizations of polymetric views⁴, visualizations of software enriched with information such as software metrics and other source code semantics. According to the authors, polymetric views help to understand the structure and detect problems of a software system in the initial phases of a reverse engineering process.

Another work targeting the comprehension of object-oriented systems was conducted by *Pacione* [Pac04], where he proposes a visualization model

⁴Polymetric Views are visualizations of a graph enriched with metrics.

based on multiple levels of abstraction, multiple facets, and/or the integration of static and dynamic information facilities. The model include a five-level abstraction scale for software comprehension. These levels are:

1. The first level (microscopic), focus on intra-object interaction and activity diagram — behavior facet.
2. The second level focus on inter-class structure and class diagram — structure facet, and inter-object interaction, object diagram — behavior facet.
3. The third level focus on system architecture and component diagram — structure facet, and component interaction, reflection model — behavior facet.
4. The fourth level focus on system deployment and deployment diagram — structure facet.
5. The fifth level focus on business behavior and use case diagram — behavior facet.

Designed to speed program comprehension, SHriMP tool (that stands for *Simple Hierarchical Multi-Perspective*) uses multiple views of software architecture [SM95]. It can be considered both a tool and a technique. It was designed to enhance the visualization and exploration of software architectures. SHriMP currently has three main applications:

- Jambalaya: a plugin created for the Protégé tool (an ontology editor), that uses SHriMP to visualize ontologies and knowledge bases.
- Creole: an Eclipse plugin that uses SHriMP to explore Java code visually.
- Stand-Alone SHriMP: a stand-alone Java application that visualizes *graph based data formats* such as GXL, RSF, XML, XMI.

Buchsbaum et al [BCH⁺01] examined the effectiveness of visualization techniques as an aid to software infrastructure comprehension at industrial level. Their results help to formulate the general requirements for software visualization and program comprehension tools.

5.4.2 Software Evolution

The visualization of the evolution of a system can be attained, at least partially, through visualizing its version history. This typically involves visualizing metrics such as number of lines of code in a particular version of a file included in the system, the percentage of growth and the percentage of change in the system, and also change complexity measures [GME05].

A relevant work in version history visualization is presented by *Gall et al* [GJR99]. They use color and third dimension techniques to visualize software release histories. Some of metrics that they visualize include size of the system in terms of lines of code, age in terms of version numbers, and error-proneness in terms of *defect density*⁵.

Their system is composed by three main entities:

- Time: the visualization is grouped according to the release sequence number. A snapshot of the system at the time of each release enables the end user to see the evolution of files between releases.
- Structure: the system is decomposed into subsystems, that in their turn are decomposed into modules and each modules comprises the source code files.
- Attributes: include version number, size, change complexity and defect density.

The kind of technology they used to build their system comprises Java and the Virtual Reality Modeling Language (VRML) to render and navigate the 3D spaces. The system offers features to the end user navigate through the visualization and extract information about the structure of the entire system for a release or focus on a particular subsystem.

Lanza and Ducasse [DL05, DLB04] also study the evolution of classes in a system using a combination of SV and software metrics. The kind of visualization provided is 2D, with rows representing the classes in a system and columns denoting the version of the system. The first column would represent version 1 of the system, the second version 2, and so on. The number of methods in the class decides the width of each rectangle representing a class, while the number of instance variables in the classes decides the height of the rectangle. This metaphor allows easy visualization of the number of classes in the system, the most recent ones that were added to the system, and the growth or stagnation phases in the evolution of the system.

Eick et al [EGK⁺02] also studied the challenge of code's change. They propose a sequence of visualizations and visual metaphors to help engineers to understand and manage the software change process. Some of these metaphors include matrix views, cityscapes, bar and pie charts, data sheets, and networks. Multiple views are combined to form perspectives that both enable discovery of high-level structure in software change data and allow effective access to details of those data.

Caudwell [Cau10] proposed a visualization system, *Gource*, to show the development history of software projects as an interactive animation. Software development is displayed by *Gource* as an animated tree with the root

⁵Defect Density is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the size of the software/-component.

directory of the project at its center. Directories appear as branches with files as leaves. Developers can be seen working on the tree at the times they contributed to the project. This project is still in development and is available at <http://code.google.com/p/gource/>.

D'Ambros [D'A10] also proposed recently a tool, **Commit 2.0**, to generate visualizations of the performed changes at different granularity levels, and let the user enrich them with annotations. The basis of his tool is based on the argument that developers write commit comments to document changes and as a means to communicate with the rest of the development team. The commit-related data contained in software repositories supports software evolution and reverse engineering activities. Originally the tool was developed for the **SmallTalk Pharo IDE** but currently it is developed for **Eclipse IDE** by Roberto Minelli⁶.

A number of software projects use versioning systems, also known as Software Configuration Management tools (SCM), such as **SVN** and **CVS**, to handle code evolution. Developers use SCMs as means of synchronization among them and to document changes through commit comments. Currently, there is a movement to develop tools for visualization of versioning systems. Some of these tools are: **CVSViewer3D** [XPM06] (a tool which extracts, processes and displays information from CVS repositories); **Chronia** [GKSD05] (uses the mapping between the changes and the author identifiers from CVS log-files to define a measurement of code ownership); **CVS-grab** [VRD04] an open framework for visual data mining of CVS repositories; and so on.

5.4.3 Visualization of Program Slices

In [BE94], *Ball et al* present **SeeSlice**, an interactive browsing tool to better visualize the data generated by slicing tools. The **SeeSlice** interface facilitates the visualization by making slices fully visible to user, even if they cross the boundaries of procedures and files.

In [GO97], *Gallagher et al* propose an approach in order to reduce the visualization complexity by using decomposition slices (see subsection 4.4.2, on page 4.4.2). The decomposition slice visualization implemented in **Surgeon's Assistant** [Gal96] visualizes the inclusion hierarchy as a graph using the **VCG** (Visualization of Compiler Graphs) [San95].

In [DKN01], *Deng et al* present **Program Slice Browser**, an interactive and integrated tool which main goal is to extract useful information from a complex program slice. Some of the features of such tool are: adaptable layout for convenient display of a slice; multi-level slice abstractions; integration with other visualization components, and capabilities to support interaction and cross-referencing within different views of the software.

⁶<http://atelier.inf.usi.ch/minellir/commit20.html>

In [Kri04], *Krinke* presents a declarative approach to layout Program Dependence Graphs (PDG) that generates comprehensible graphs of small to medium size procedures. The authors discussed how a layout for PDG can be generated to enable an appealing presentation. The PDG and the computed slices are shown in a graphical way. This graphical representation is combined with the textual form, as the authors argue that is much more effective than the graphical one. The authors also solved the problem of loss of locality in a slice, using a distance-limited approach. With their approach, they aimed at answering the following questions: 1) why a statement is included in the slice?, and 2) how strong is the influence of the statement on the criterion?

In [Bal01], *Balmas* presents an approach to decompose System Dependence Graphs in order to have graphs of manageable size: groups of nodes are collapsed into one node. The system implemented provides three possible decompositions to be browsed and analyzed through a graphical interface: nodes belonging to the same procedure; nodes belonging to the same loop; nodes belonging to the two previous ones.

5.4.4 Algorithms Visualization/Animation

Algorithm visualization/animation can both help instructors to explain and learners to understand algorithms, software engineering principles and practices [Hun02].

Usually, algorithm animation involves to show the values of parameters and variables at a certain state, and a visual representation of the objects being animated. Details can be shown at different levels of abstraction, omitting in each analysis/comprehension step the irrelevant details. Smooth transitions between different states can make it easier to follow the way how the algorithm works through graphical representations of data structures. Graphs and Abstract Syntax Trees are common data structures used in algorithm animation.

Baloian et al [BBL05] proposed an approach to developing algorithm visualization that seeks to construct context-independent interfaces allowing the learner to interactively control and explore the actions in the algorithm. The proposed approach replaces standard control using mouse clicks on graphical displays with a concept called Concept Keyboards (CK). Instead of using the traditional keyboard (which is mostly used to process text and each key is labeled with a specific character or number), they use a customized keyboard with a reduced number of keys targeting specifically the control of algorithm visualization and animation.

ALMA [dCHP07] is a general purpose visualizer and animator that takes as input an abstract representation (a decorated Abstract Syntax Tree). Program execution is simulated step-by-step by rewriting the tree, which is displayed after each step to produce an animation. The tool relies upon

a dedicated front-end that converts input programs of a concrete programming language into the generic internal representation. **ALMA**² (previously referred) is its successor and improves it by adding information about the problem domain.

Balsa [BS84] was one of the first algorithm animation systems. This is an interactive animation system that supports multiple simultaneous views over data structures handled by one or more algorithms, providing the capability to display multiple algorithms executing simultaneously.

Tango [Sta90] introduced the *path-transition paradigm* for animation design using smooth animations. **Tango** visualizations are seen as mappings from program events to animation actions. The occurrence of an event will trigger a smooth animation of one or more graphical objects.

Zeus [Bro92] was also one of the first major interactive software visualization systems. It supports multiple synchronized views and allows users to edit those views, change a data item representation, or control the program execution. **Zeus** is a multi-threaded system, which makes it appropriate for animating parallel programs.

Leonardo [CDFP00] is an integrated environment that allows the user to edit, compile, execute, and animate general-purpose C programs. **Leonardo** automatically detects visual events during the execution of programs and optimizes the creation of visualizations following an incremental approach.

The **Animal** [RSF00] algorithm animator system allows the user to jump to any point within the animation in either direction. It provides three ways of generating animations: using scripting, using a **Java**-based API, or using a GUI.

Jawaa [PR98] is an algorithm visualization system that offers general graphic primitives and effects. The user can view the animation as a slide show without breaks between steps or on a step-by-step basis.

Other applications of software visualization include software security [CA04, Yoo04, Ma04, YVQ⁺10], software architectures [FDJ98, SB08, SFFG10], data mining [Kei02, BDW05] and fault localization [RCB⁺03].

Chapter 6

Verification Graphs

A fact is a simple statement that everyone believes. It is innocent, unless found guilty. A hypothesis is a novel suggestion that no one wants to believe. It is guilty, until found effective.

Edward Teller, 1908 — 2003

Every program implicitly asserts a theorem to the effect that if certain input conditions are met then the program will do what its specification or documentation says it will. Making that theorem true is not merely a matter of luck or patient debugging; writing a correct program can be greatly aided by a logical analysis of what it is supposed to do. Including the specification in the form of annotations within the program has been a successful approach in part because it makes the verification process easier.

This idea of including information (specifications) about how the program should behave, was born in the eighties the *Design-by-Contract* (DbC) software development method. DbC [Mey86, Mey92] *is a metaphor on how elements of a software system collaborate with each other, on the basis of mutual obligations and benefits. The metaphor comes from business life, where a “client” and a “supplier” agree on a “contract”*.¹

In terms of programming, this means that the programmer (the “supplier”) should include in their code the “client” requirements. Usually this is done through the use of *preconditions*, *postconditions* and *invariants*. These annotations spread over the code represent the “contract”. The contract for a software component can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. In terms of verification terminology, a contract for a component is simply a pair con-

¹Retrieved from http://www.eiffel.com/developers/design_by_contract.html in May, 2011.

sisting of a precondition and a postcondition. It certifies the results that can be expected after execution of the component, but it also constrains the input values of the component.

In this context of including the contracts in the code, the broad adoption of annotation languages for the major programming languages reinforces the perceived importance of using this approach of DbC, which facilitates modular verification and certified code reuse. These annotation languages include for instance the Java Modeling Language (JML) [BCC⁺05]; Spec# [BRLS04], a formal language for C# API contracts; and the ANSI/ISO C Specification Language (ACSL) [BCF⁺10]. The reader is referred to [HLL⁺09] for a fairly recent survey on specification languages.

As was discussed in Chapter 3, one way to verify the correctness of a program is through the use of Verification Condition Generators (VCGens). These VCGens read a piece of code together with a specification and compute a set of verification conditions (VCs) that are then sent to a theorem prover. If all the verification conditions of a program are correct, then the program is said to be correct. This approach can of course be used to establish the correctness of the code with respect to the contracts embedded in it.

However, when using these VCGens, it may be hard to establish a connection between the set of generated VCs and the source code. Thus, in case one of these VCs be incorrect, the discovery of which statements (or which execution paths) are leading to the incorrectness of the program may be impossible.

For instance, consider the fragment of a program used to calculate the income taxes in the United Kingdom² (method `TaxesCalculation` in Listing 6.1). Consider as precondition $P = \text{age} \geq 18$ and as postcondition $Q = \text{personal} > 5750$. A careful observation of the method allows one to detect problems, as in some circumstances the result value of *personal* can in fact be less than 5750. If one decides to check whether the program is correct or not using a standard VCGen algorithm like the one presented in subsection 3.2.3, the result is a single VC, too big and complex to be understandable (the resulting verification condition is shown in Figure 6.1). It could be hard to understand what is causing the incorrectness of the program just by looking at the VC and the output of the prover.

In this chapter we introduce an alternative way to verify if a program is correct through the use of Control Flow Graphs (Definition 2 on page 9). In particular, the properties of labeled CFGs (CFGs whose edges have labels corresponding to assertions propagated from the specification of a given block) are studied in the context of contract-annotated procedures. It is shown how these graphs can be used for generating verification conditions in a way that:

²The complete source code of this program can be found in [HHF⁺02] and has been used as a benchmark test for slicing algorithms based on assertions.

```

2  if (age >= 75) then
    personal := 5980
4  else
    if (age >= 65) then
        personal := 5720
6    else
        personal := 4335
8
10 if ((age >= 65) and (income > 16800)) then
    t := personal - ((income - 16800)/2)
12 else
    if (t > 4335) then
        personal := t + 2000
14    else
        personal := 4335

```

Listing 6.1: Program TaxesCalculation

$$\begin{aligned}
& ((age \geq 18) \rightarrow (((age \geq 75) \rightarrow (((age \geq 65) \wedge (income > 16800)) \rightarrow (((5980 - \\
& ((income - 16800)/2)) > 4335) \rightarrow (((5980 - ((income - 16800)/2)) + 2000) > 5750)) \\
& \wedge (!((5980 - ((income - 16800)/2)) > 4335)) \rightarrow (4335 > 5750)))) \wedge (!((age \geq 65) \\
& \wedge (income > 16800))) \rightarrow true))) \wedge (!((age \geq 75)) \rightarrow (((age \geq 65) \rightarrow (((age \geq 65) \\
& \wedge (income > 16800)) \rightarrow (((5720 - ((income - 16800)/2)) > 4335) \rightarrow (((5720 - \\
& ((income - 16800)/2)) + 2000) > 5750)) \wedge (!((5720 - ((income - 16800)/2)) > 4335)) \\
& \rightarrow (4335 > 5750)))) \wedge (!((age \geq 65) \wedge (income > 16800))) \rightarrow true))) \wedge (!((age \geq 65)) \\
& \rightarrow (((age \geq 65) \wedge (income > 16800)) \rightarrow (((4335 - ((income - 16800)/2)) > 4335) \\
& \rightarrow (((4335 - ((income - 16800)/2)) + 2000) > 5750)) \wedge (!((4335 - ((income - 16800)/2)) \\
& > 4335)) \rightarrow (4335 > 5750)))) \wedge (!((age \geq 65) \wedge (income > 16800))) \rightarrow true))))))
\end{aligned}$$

Figure 6.1: Verification condition for the UKTakesCalculation program

1. combines forward propagation of preconditions and backward-propagation of postconditions, thus bringing together the advantages of forward and backward reasoning;
2. allows for a closer relation with error paths;
3. can be implemented either based on pre-defined strategies or interactively (user-guided VC generation);
4. lends itself to visualization applications.

Structure of the chapter. Section 6.1 recalls some basic definitions and introduces the notation that will be used hereafter. Properties about the verification conditions generated using weakest precondition or strongest postcondition propagation are discussed and it is shown how both strategies can be combined. Finally, the setting necessary to consider a total correctness scenario is introduced. In Section 6.2, the definition of Labeled Control Flow Graph and its construction is introduced. In Section 6.3, the notion of

Verification Graph is presented and the process of verifying a program in an interactive way through the use of these graphs is discussed. An example is also presented.

6.1 Verification Conditions

To illustrate the ideas that will be presented, recall the language defined in Figure 3.3 on page 58. Its syntax is defined in two levels: first blocks (or sequences) of commands are formed, which can be seen as standard programs of a *While* programming language. Then procedures consisting of a block of code annotated with a precondition and a postcondition (that form the procedure's specification, or *contract*) are constructed. Commands include a procedure call in addition to **skip**, assignment, conditional branching and loops.

Occurrences of variables in the precondition and postcondition of a procedure refer to their values in the pre-state and post-state of execution of the procedure respectively. Each block is additionally annotated with loop invariants. The language of annotated assertions (used for writing invariants and contracts) extends boolean expressions with implication and first-order quantification. Defining the syntax of assertions as a superset of boolean expressions is a common practice in specification languages based on contracts (such as SPARK [Bar03]), since it facilitates the task of writing annotated code.

A *program* is then a non-empty set of (mutually recursive) procedure definitions³. For the sake of simplicity only *parameterless procedures* will be considered, that share a set of global variables, but the ideas presented here can be adapted to cope with parameters (passed by value or by reference), as well as return values of functions. Note that a program defined in this way is close to the notion of class in object-oriented programming, with procedures and global variables playing the role of methods and class attributes. But of course other notions like the creation of class instances or inheritance are absent from this analogy.

A program is well-formed if the name of every procedure defined in it is unique and the program is closed with respect to procedure invocation. We will write $\mathbf{PN}(\Pi)$ for the set of names of procedures defined in program Π . The operators **pre**, **post**, and **body** will be used to refer to a procedure's precondition, postcondition, and body command, respectively, i.e. given the procedure definition **pre** P **post** Q **proc** $\mathbf{p} = S$ with $\mathbf{p} \in \mathbf{PN}(\Pi)$, one has $\mathbf{pre}_\Pi(\mathbf{p}) = P$, $\mathbf{post}_\Pi(\mathbf{p}) = Q$, and $\mathbf{body}_\Pi(\mathbf{p}) = S$. The program name will be omitted when clear from context.

³Operationally, an entry point would have to be defined for each such program, but that is not important for our current purpose.

Weakest Preconditions and Strongest Postconditions. Recall from Section 3.2 that one way to obtain a verification condition for a program S (a formula whose validity implies the partial correctness of S with respect to a specification consisting of precondition P and postcondition Q) is to use Dijkstra’s *weakest liberal precondition* [Dij76a] predicate transformer: $wlp.S.Q$ designates the weakest precondition that will lead to Q being true in the final state, *if the execution of S terminates*. The verification condition for S to meet its specification can then be written as $P \rightarrow wlp.S.Q$. In this work it is assumed that every loop is annotated with a user-provided *invariant*; in the presence of a loop invariant, the different required conditions (the invariant is initially true, it is preserved by loop iterations, and upon termination of the loop it is stronger than the desired postcondition) can be combined in a single formula to give the weakest precondition of each loop. This requires the use of universal quantifiers over state variables, to isolate the different conditions.

Following Section 3.2, we use a different approach that produces a set of independent verification conditions, avoiding the introduction of quantifiers. This approach requires calculating a notion of precondition that is related to wlp , but differs in that the weakest precondition of a loop is simply defined as being its invariant, regardless of whether termination is guaranteed or not. Figure 6.2 recalls the definition of the function $wprec$ corresponding to this notion. Figure 6.2 also contains the definition of the function $spost$, corresponding to the symmetric notion of *strongest postcondition* that will be true in the final state of the program S when its execution starts in a state satisfying P . Throughout the remaining chapters whenever the weakest precondition or strongest postcondition of a program is referred, it is meant the conditions calculated by these functions.

A final remark: the definitions of weakest precondition and strongest postcondition for the procedure call command in Figure 6.2 take into account the *adaptation* between the procedure’s contract and the postcondition/precondition required in the context in which the procedure is called. In the absence of auxiliary variables, the following would be sufficient:

$$wprec(\text{call } p, Q) = \mathbf{pre}(p) \qquad \qquad \text{spost}(\text{call } p, P) = \mathbf{post}(p) \qquad (6.1)$$

$$\text{VC}^w(\text{call } p, Q) = \{\mathbf{post}(p) \rightarrow Q\} \qquad \text{VC}^s(\text{call } p, P) = \{P \rightarrow \mathbf{pre}(p)\} \qquad (6.2)$$

But of course auxiliary variables are essential, since they allow to specify how the output of a procedure is related to its input. In their presence a single verification condition should be generated for each call; it is no longer possible to separate $\mathbf{pre}(p)$ and $\mathbf{post}(p)$ as above. This makes it necessary to rename and universally quantify program variables occurring in one of the pre-state or post-state, to distinguish occurrences of the same variable in the two states – a syntactic treatment of states which is avoided in the remaining clauses of the VCGen. On the other hand auxiliary variables are

$$\begin{aligned}
& \text{wprec}(\text{skip}, Q) = Q \\
& \text{wprec}(x := e, Q) = Q[e/x] \\
& \text{wprec}(\text{if } b \text{ then } S_t \text{ else } S_f, Q) = (b \rightarrow \text{wprec}(S_t, Q)) \wedge (\neg b \rightarrow \text{wprec}(S_f, Q)) \\
& \text{wprec}(\text{while } b \text{ do } \{I\} S, Q) = I \\
& \text{wprec}(\text{call } \mathbf{p}, Q) = \forall \bar{x}_f. (\forall \bar{y}_f. \text{pre}(\mathbf{p})[\bar{y}_f/\bar{y}] \rightarrow \text{post}(\mathbf{p})[\bar{y}_f/\bar{y}, \bar{x}_f/\bar{x}]) \rightarrow Q[\bar{x}_f/\bar{x}] \\
& \text{wprec}(C; S, Q) = \text{wprec}(C, \text{wprec}(S, Q)) \\
\\
& \text{VC}^w(\text{skip}, Q) = \emptyset \\
& \text{VC}^w(x := e, Q) = \emptyset \\
& \text{VC}^w(\text{if } b \text{ then } S_t \text{ else } S_f, Q) = \text{VC}^w(S_t, Q) \cup \text{VC}^w(S_f, Q) \\
& \text{VC}^w(\text{while } b \text{ do } \{I\} S, Q) = \{I \wedge b \rightarrow \text{wprec}(S, I), I \wedge \neg b \rightarrow Q\} \cup \text{VC}^w(S, I) \\
& \quad = \{I \wedge \neg b \rightarrow Q\} \cup \text{VCG}^w(I \wedge b, S, I) \\
& \text{VC}^w(\text{call } \mathbf{p}, Q) = \emptyset \\
& \text{VC}^w(C; S, Q) = \text{VC}^w(C, \text{wprec}(S, Q)) \cup \text{VC}^w(S, Q) \\
\\
& \text{spost}(\text{skip}, P) = P \\
& \text{spost}(x := e, P) = \exists v. P[v/x] \wedge x = e[v/x] \\
& \text{spost}(\text{if } b \text{ then } S_t \text{ else } S_f, P) = \text{spost}(S_t, b \wedge P) \vee \text{spost}(S_f, \neg b \wedge P) \\
& \text{spost}(\text{while } b \text{ do } \{I\} S, P) = I \wedge \neg b \\
& \text{spost}(\text{call } \mathbf{p}, P) = \exists \bar{x}_f. P[\bar{x}_f/\bar{x}] \wedge (\forall \bar{y}_f. \text{pre}(\mathbf{p})[\bar{y}_f/\bar{y}, \bar{x}_f/\bar{x}] \rightarrow \text{post}(\mathbf{p})[\bar{y}_f/\bar{y}]) \\
& \text{spost}(C; S, P) = \text{spost}(S, \text{spost}(C, P)) \\
\\
& \text{VC}^s(\text{skip}, P) = \emptyset \\
& \text{VC}^s(x := e, P) = \emptyset \\
& \text{VC}^s(\text{if } b \text{ then } S_t \text{ else } S_f, P) = \text{VC}^s(S_t, P) \cup \text{VC}^s(S_f, P) \\
& \text{VC}^s(\text{while } b \text{ do } \{I\} S, P) = \{P \rightarrow I, \text{spost}(S, I \wedge b) \rightarrow I\} \cup \text{VC}^s(S, I \wedge b) \\
& \quad = \{P \rightarrow I\} \cup \text{VCG}^s(I \wedge b, S, I) \\
& \text{VC}^s(\text{call } \mathbf{p}, P) = \emptyset \\
& \text{VC}^s(C; S, P) = \text{VC}^s(C, P) \cup \text{VC}^s(S, \text{spost}(C, P)) \\
\\
& \text{where } \bar{y} \text{ is a sequence of the auxiliary variables of } \mathbf{p} \\
& \quad \bar{x} \text{ is a sequence of the program variables occurring in } \text{body}(\mathbf{p}) \\
& \quad \bar{x}_f \text{ and } \bar{y}_f \text{ are sequences of fresh variables} \\
& \quad \text{The expression } t[\bar{e}/\bar{x}], \text{ with } \bar{x} = x_1, \dots, x_n \text{ and } \bar{e} = e_1, \dots, e_n, \\
& \quad \text{denotes the parallel substitution } t[e_1/x_1, \dots, e_n/x_n]
\end{aligned}$$

Figure 6.2: Verification conditions for blocks of commands with procedure call

local to a procedure and have also to be renamed to avoid capture.

Our definitions adapt to our context the rule proposed by Kleymann [Kle99] as an extension to Hoare logic. The reader is referred to that paper for details and a discussion of different approaches to adaptation.

To sum up, there are two well-established methods for producing sets of verification conditions, based respectively on *weakest precondition* and *strongest postcondition* calculations. Verifying the behavior of programs is of course in general an undecidable problem, and as such automation is necessarily limited. Typical tools require the user to provide additional information, in particular *loop invariants*, and one could be tempted to think that the only interesting problem of program verification is automating the generation of invariants: if appropriate annotations are provided, then generating and proving verification conditions should be straightforward.

The Case for Forward Propagation. The forward propagation of assertions was a key part of Floyd's inductive assertion method for reasoning about programs [Flo67]. Strongest postcondition calculations also propagate assertions forward, and have been repeatedly advocated as a tool to assist in different software engineering tasks [PD95, GC95]. Mike Gordon [GC10] has more recently argued for the advantages of forward reasoning, pointing out the similarities between strongest postcondition calculations and *symbolic execution*, a method for analyzing and checking properties of programs based on simulating their execution with symbolic input values [PV09]. An account of verification conditions based on forward propagation provides an interesting link with techniques based on symbolic execution.

A more recent work [JB10] presents another argument for the use of forward propagation: in some situations it allows for the verification conditions to be simplified while they are computed, using standard optimizations like constant propagation and even dead code elimination.

Consider for instance the program $x := 10; y := 5 * x$. Its VCs can be calculated using instead the program $x := 10; y := 50$. And for the program $x := 10; y := 5 * x; \text{ if } y > 0 \text{ then } S_1 \text{ else } S_2$, they can be calculated instead from $x := 10; y := 50; S_1$. The resulting VCs are significantly simpler.

The resurgent use of strongest postconditions in a number of recent papers is also due to the fact that it is now known how to compute them in a way that is uncluttered by existential quantifiers, and moreover produces VCs that are at least as small as those generated for the same program using backward propagation (and arguably smaller, if simplifications such as mentioned above are carried out).

Verification Conditions for Partial Correctness. As discussed in Chapter 3, in this thesis verification conditions are calculated based on Hoare logic [Hoa69] rather than guarded commands and predicate transformers. In

this approach the verification conditions required for the partial correctness of the program S with respect to specification (P, Q) are the side conditions of a derivation (or proof tree) of the logic. If the verification conditions are all valid, then it is possible to construct a derivation with the Hoare triple $\{P\} S \{Q\}$ as conclusion, in which case S is (partially) correct.

The derivations with a given conclusion are not unique; although they do not need to be explicitly constructed in order for the side conditions to be obtained, some strategy is still necessary to direct the process. In this chapter two such strategies will be used, based on weakest preconditions and on strongest postconditions respectively. Technically, these strategies are responsible for selecting intermediate conditions for the *sequence* rule of Hoare logic: when considering a derivation for the triple $\{P\} S_1 ; S_2 \{Q\}$, this rule states that two derivations should be recursively considered, for the triples $\{P\} S_1 \{R\}$ and $\{R\} S_2 \{Q\}$ for some condition R . Our first strategy sets R to be $\text{wprec}(S_2, Q)$; the second strategy sets R to be $\text{spost}(S_1, P)$.

Each of these strategies results in a different set of verification conditions, as previously stated in Definitions 50 and 52 of Chapter 3:

$$\begin{aligned} \text{VCG}^w(P, S, Q) &= \{P \rightarrow \text{wprec}(S, Q)\} \cup \text{VC}^w(S, Q) \\ &\text{and} \\ \text{VCG}^s(P, S, Q) &= \text{VC}^s(S, P) \cup \{\text{spost}(S, P) \rightarrow Q\} \end{aligned}$$

where the functions VC^w and VC^s are also defined in Figure 6.2. These auxiliary functions are responsible for traversing the implicit derivations and collecting the side conditions along the way. These traversals are based uniquely on one of the conditions given in the specification (the postcondition and precondition respectively); an additional formula involving the other condition must then be added to this set.

Finally, the generation of verification conditions should of course be *sound* with respect to the operational semantics of the language: if they are all valid then this should constitute a guarantee that the program is indeed correct with respect to its specification (P, Q) :

If either $\models \text{VCG}^w(P, S, Q)$ or $\models \text{VCG}^s(P, S, Q)$ and S is executed in a state that satisfies P , then if S terminates the final state satisfies Q .

This is easy to prove for such a simple language, with respect to a standard evaluation semantics. The reader is directed to [FP11] for a proof, and also for more details on verification conditions and their relation to Hoare logic and Dijkstra's predicate transformers.

A correspondence result between both strategies can be stated as follows.

Lemma 1. *For every precondition P , postcondition Q , and program S ,*

$$\models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad \models \text{VCG}^s(P, S, Q)$$

Proof. By induction on the structure of S . For the case where S is

while b **do** $\{I\} S_b$

the following is used as induction hypothesis: $\models \text{VCG}^w(I \wedge b, S_b, I)$ iff $\models \text{VCG}^s(I \wedge b, S_b, I)$. \square

Since the language under consideration has integer variables only, verifying the correctness of programs can be achieved by applying the VCGen and exporting the resulting proof obligations to a proof tool capable of reasoning with integer arithmetics.

Observe that this is not a fully automated method since it requires users to provide the annotations (recall the discussion about the automated and the manual/semi-automatic techniques in Chapter 3). Furthermore, undecidability of first-order logic means that interactive proof may be necessary⁴, but it must also be noted that the power of automatic proof has progressed significantly in recent years. Real-language implementations of many standard algorithms can now be proved fully automatically, which is certainly a great advance with respect to what could be achieved, say, ten years ago. Recent approaches build in particular on advances in SMT solvers⁵ (that combine useful programming theories), and also on combinations of automatic provers (for the easy proofs) and interactive proof assistants (for the hard parts).

Combining Backward and Forward Propagation. As mentioned before, the set of VCs given by Definitions 50 and 52 is not unique. A more general method, which allows us to verify the correctness of a given block of code using weakest preconditions for part of the block and strongest postconditions for the remaining commands, will now be considered.

Let $S = C_1 ; \dots ; C_n$, $1 \leq k \leq n$. A dedicated notation will be used for the weakest precondition of a suffix of S and the strongest postcondition of a prefix of S , as well as for the (partial correctness) verification conditions of both, as follows.

⁴In the scope of program verification, failure of automatic proof does not mean a program is not correct, it just means that interactive proof should be used instead to clarify whether a given proof obligation is indeed invalid or not.

⁵A brief introduction to SMT is given in Appendix B.

$$\begin{aligned}
\overline{\text{wprec}}_k(S, Q) &= \text{wprec}(C_k; C_{k+1}; \dots; C_n, Q) \\
\overline{\text{wprec}}_{n+1}(S, Q) &= Q \\
\overline{\text{VC}}_k^w(S, Q) &= \text{VC}^w(C_k; C_{k+1}; \dots; C_n, Q) \\
\overline{\text{VC}}_{n+1}^w(S, Q) &= \{\} \\
\overline{\text{spost}}_0(S, P) &= P \\
\overline{\text{spost}}_k(S, P) &= \text{spost}(C_1; \dots; C_{k-1}; C_k, P) \\
\overline{\text{VC}}_k^s(S, P) &= \text{VC}^s(C_1; \dots; C_{k-1}; C_k, P) \\
\overline{\text{VC}}_0^s(S, P) &= \{\}
\end{aligned}$$

Then, it is defined:

$$\begin{aligned}
\text{VCG}^k(P, S, Q) &= \overline{\text{VC}}_k^s(S, P) \cup \{\overline{\text{spost}}_k(S, P) \rightarrow \overline{\text{wprec}}_{k+1}(S, Q)\} \\
&\cup \overline{\text{VC}}_{k+1}^w(S, Q)
\end{aligned}$$

Note that $\text{VCG}^w(P, S, Q) = \text{VCG}^0(P, S, Q)$; we also define the following, entirely based on strongest postconditions:

$$\text{VCG}^s(P, S, Q) = \text{VCG}^n(P, S, Q)$$

The following lemma states some facts about the combined generation of VCs. It implies that we can equivalently use any value of k to generate verification conditions.

Lemma 2. *Let (P, Q) be a specification and $S = C_1; \dots; C_n$ a program.*

1. *for $k \in \{0, \dots, n\}$,*

$$\begin{aligned}
\models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_k^s(S, P), \overline{\text{spost}}_k(S, P) \rightarrow \overline{\text{wprec}}_{k+1}(S, Q), \\
& \overline{\text{VC}}_{k+1}^w(S, Q),
\end{aligned}$$

2. *If $C_k = \text{if } b \text{ then } S_t \text{ else } S_f$ for some $k \in \{1, \dots, n\}$, then*

$$\begin{aligned}
\models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_{k-1}^s(S, P), \\
& \text{VCG}^w(\overline{\text{spost}}_{k-1}(S, P) \wedge b, S_t, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \text{VCG}^w(\overline{\text{spost}}_{k-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \overline{\text{VC}}_{k+1}^w(S, Q)
\end{aligned}$$

3. If $C_k = \mathbf{while} \ b \ \mathbf{do} \ \{I\} \ S_b$ for some $k \in \{1, \dots, n\}$, then

$$\begin{aligned} \models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow I, \\ & \text{VCG}^w(I \wedge b, S_b, I), \\ & I \wedge \neg b \rightarrow \overline{\text{wprec}}_{k+1}(S, Q), \\ & \overline{\text{VC}}_{k+1}^w(S, Q) \end{aligned}$$

Proof. 1. Applying repeatedly the definitions of wprec , VC^w , spost , VC^s , and Lemma 1:

$$\begin{aligned} & \models P \rightarrow \text{wprec}(S, Q), \text{VC}^w(S, Q) \\ \text{iff} \quad & \models P \rightarrow \text{wprec}(C_1, \text{wprec}(C_2; \dots; C_n, Q)), \\ & \text{VC}^w(C_1, \text{wprec}(C_2; \dots; C_n, Q)), \text{VC}^w(C_2; \dots; C_n, Q) \\ \text{iff} \quad & \models \text{VC}^s(C_1, P), \text{spost}(C_1, P) \rightarrow \text{wprec}(C_2; \dots; C_n, Q), \\ & \text{VC}^w(C_2; \dots; C_n, Q) \\ \text{iff} \quad & \models \text{VC}^s(C_1, P), \text{spost}(C_1, P) \rightarrow \text{wprec}(C_2, \text{wprec}(C_3; \dots; C_n, Q)), \\ & \text{VC}^w(C_2, \text{wprec}(C_3; \dots; C_n, Q)), \\ & \text{VC}^w(C_3; \dots; C_n, Q) \\ \text{iff} \quad & \models \text{VC}^s(C_1, P), \text{VC}^s(C_2, \text{spost}(C_1, P)), \\ & \text{spost}(C_2, \text{spost}(C_1, P)) \rightarrow \text{wprec}(C_3; \dots; C_n, Q), \\ & \text{VC}^w(C_3; \dots; C_n, Q) \\ \text{iff} \quad & \models \text{VC}^s(C_1; C_2, P), \\ & \text{spost}(C_1; C_2, P) \rightarrow \text{wprec}(C_3; \dots; C_n, Q), \\ & \text{VC}^w(C_3; \dots; C_n, Q) \\ & \dots \\ \text{iff} \quad & \models \text{VC}^s(C_1; \dots; C_k, P), \\ & \text{spost}(C_1; \dots; C_k, P) \rightarrow \text{wprec}(C_{k+1}; \dots; C_n, Q), \\ & \text{VC}^w(C_{k+1}; \dots; C_n, Q) \end{aligned}$$

2. Using the definition of the VCGen and Lemma 2 (1), one has the following

$$\begin{aligned} & \models \text{VCG}^w(P, S, Q) \\ \text{iff} \quad & \models \overline{\text{VC}}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow \overline{\text{wprec}}_k(S, Q), \overline{\text{VC}}_k^w(S, Q) \\ \text{iff} \quad & \models \overline{\text{VC}}_{k-1}^s(S, P), \\ & \overline{\text{spost}}_{k-1}(S, P) \rightarrow (b \rightarrow \text{wprec}(S_t, \overline{\text{wprec}}_{k+1}(S, Q))) \wedge \\ & (\neg b \rightarrow \text{wprec}(S_f, \overline{\text{wprec}}_{k+1}(S, Q))), \\ & \text{VC}^w(S_t, \overline{\text{wprec}}_{k+1}(S, Q)), \text{VC}^w(S_f, \overline{\text{wprec}}_{k+1}(S, Q)), \overline{\text{VC}}_{k+1}^w(S, Q) \end{aligned}$$

$$\begin{aligned}
& \text{iff } \models \overline{VC}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \wedge b \rightarrow \text{wprec}(S_t, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \quad \text{VC}^w(S_t, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \quad \overline{\text{spost}}_{k-1}(S, P) \wedge \neg b \rightarrow \text{wprec}(S_f, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \quad \text{VC}^w(S_f, \overline{\text{wprec}}_{k+1}(S, Q)), \overline{VC}_{k+1}^w(S, Q) \\
& \text{iff } \models \overline{VC}_{k-1}^s(S, P), \text{VCG}^w(\overline{\text{spost}}_{k-1}(S, P) \wedge b, S_t, \overline{\text{wprec}}_{k+1}(S, Q)), \\
& \quad \text{VCG}^w(\overline{\text{spost}}_{k-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{k+1}(S, Q)), \overline{VC}_{k+1}^w(S, Q)
\end{aligned}$$

3. We reason as follows, again using the definition of the VCGen and Lemma 2 (1)

$$\begin{aligned}
& \models \text{VCG}^w(P, S, Q) \\
& \text{iff } \models \overline{VC}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow \overline{\text{wprec}}_k(S, Q), \overline{VC}_k^w(S, Q) \\
& \text{iff } \models \overline{VC}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow I, I \wedge b \rightarrow \text{wprec}(S_b, I), \\
& \quad I \wedge \neg b \rightarrow \overline{\text{wprec}}_{k+1}(S, Q), \text{VC}^w(S_b, I), \overline{VC}_{k+1}^w(S, Q) \\
& \text{iff } \models \overline{VC}_{k-1}^s(S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow I, \text{VCG}^w(I \wedge b, S_b, I), \\
& \quad I \wedge \neg b \rightarrow \overline{\text{wprec}}_{k+1}(S, Q), \overline{VC}_{k+1}^w(S, Q)
\end{aligned}$$

□

Verification Conditions for Total Correctness. In a total correctness setting the verification conditions are further required to grant termination of programs. In our language expression evaluation always terminates, so what is required is that every loop in a given program terminates. For this it is required that each loop contains an additional annotation, an integer expression called a *loop variant*:

$$\text{Comm} \ni C ::= \dots \mid \text{while } b \text{ do } \{A, e\} S$$

If for every loop in the program the value of the respective variant is initially non-negative and strictly decreases with each iteration, the program is guaranteed to terminate. The VCGen of Figure 6.2 can be extended to cope with total correctness by simply modifying the verification conditions of loops. The function VC_t^w (resp. VC_t^s) has the same definition as VC^w (resp. VC^s) except for the case of loops, which is given as follows for a loop annotated with invariant I and variant e_v :

$$\begin{aligned} \text{VC}_t^w(\text{while } b \text{ do } \{I, e_v\} S, Q) = & \{I \wedge b \rightarrow e_v \geq 0, I \wedge b \wedge e_v = x_0 \\ & \rightarrow \text{wprec}(S, I \wedge e_v < x_0), I \wedge \neg b \rightarrow Q\} \\ & \cup \text{VC}_t^w(S, I \wedge e_v < x_0) \end{aligned}$$

$$\begin{aligned} \text{VC}_t^s(\text{while } b \text{ do } \{I, e_v\} S, P) = & \{I \wedge b \rightarrow e_v \geq 0, P \rightarrow I, \\ & \text{spost}(S, I \wedge b \wedge e_v = x_0) \rightarrow I \wedge e_v < x_0\} \\ & \cup \text{VC}_t^s(S, I \wedge b \wedge e_v = x_0) \end{aligned}$$

Note that the weakest precondition and strongest postcondition functions wprec and spost are still defined as before. Note also the use of an auxiliary variable x_0 to store the initial value of the variant (regarding an arbitrary loop iteration), which then allows us to force the postcondition $e_v < x_0$. Now let

$$\begin{aligned} \text{VCG}_t^w(P, S, Q) &= \{P \rightarrow \text{wprec}(S, Q)\} \cup \text{VC}_t^w(S, Q) \\ &\text{and} \\ \text{VCG}_t^s(P, S, Q) &= \text{VC}_t^s(S, P) \cup \{\text{spost}(S, P) \rightarrow Q\} \end{aligned}$$

The resulting VCGens are sound with respect to total correctness, i.e.

If either $\models \text{VCG}_t^w(P, S, Q)$ or $\models \text{VCG}_t^s(P, S, Q)$ and S is executed in a state that satisfies P , then S terminates, and moreover the final state satisfies Q .

Note that it is immediate from the definitions of VC_t^w and VC^w (resp. VC_t^s and VC^s) that

$$\begin{aligned} \models \text{VCG}_t^w(P, S, Q) &\text{ implies } \models \text{VCG}^w(P, S, Q), \quad \text{and} \\ \models \text{VCG}_t^s(P, S, Q) &\text{ implies } \models \text{VCG}^s(P, S, Q) \end{aligned}$$

which is in accordance with the fact that total correctness is a stronger notion than partial correctness. In fact, in practice the total correctness of a program is often established by first proving its partial correctness and then additionally checking that it terminates on initial states satisfying the precondition.

The following lemma states that the weakest precondition and the strongest postcondition strategies are equivalent for calculating total correctness verification conditions:

Lemma 3. *For every precondition P , postcondition Q , and program S ,*

$$\models \text{VCG}_t^w(P, S, Q) \quad \text{iff} \quad \models \text{VCG}_t^s(P, S, Q)$$

Proof. By induction on the structure of S . For the case where S is

while b **do** $\{I, e_v\} S_b$

, the following is used as induction hypothesis: $\models \text{VCG}_t^w(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0)$ iff $\models \text{VCG}_t^s(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0)$. \square

6.2 Labeled Control Flow Graphs

This chapter introduces the notion of control flow graph annotated with pairs of assertions and properties of these Labeled Control Flow Graphs (LCFG) are studied. It will be explained how they can be used as a basis for the interactive generation of verification conditions. In the next chapter, another application of these graphs will be presented.

Before the formal definition of LCFG, it is necessary to introduce the notions of *subprogram* and *local specification*.

Definition 74 (Subprogram and Local Specification). *Let S be a program and (P, Q) a specification for it. \hat{S} is a subprogram of S , and (\hat{P}, \hat{Q}) is its local specification — written as $(\hat{P}, \hat{S}, \hat{Q}) \in (P, S, Q)$. The \in relation is defined inductively as follows.*

- $(P, S, Q) \in (P, S, Q)$;
- If $(P_1, S_1, Q_1) \in (P, S, Q)$ and $(P_2, S_2, Q_2) \in (P_1, S_1, Q_1)$ then $(P_2, S_2, Q_2) \in (P, S, Q)$.
- If $S = C_1 ; \dots ; C_n$ and $C_i = \text{if } b \text{ then } S_t \text{ else } S_f$ for some i with $1 \leq i \leq n$, then
 - $(\overline{\text{spost}}_{i-1}(S, P) \wedge b, S_t, \overline{\text{wprec}}_{i+1}(S, Q)) \in (P, S, Q)$
 - $(\overline{\text{spost}}_{i-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{i+1}(S, Q)) \in (P, S, Q)$
- If $S = C_1 ; \dots ; C_n$ and $C_i = \text{while } b \text{ do } \{I\} S_b$ for some i with $1 \leq i \leq n$, then $(I \wedge b, S_b, I) \in (P, S, Q)$.

As expected, subprograms of a correct program are correct with respect to their local specifications.

Lemma 4. *Let S, \hat{S} be programs and (P, Q) a specification such that $(\hat{P}, \hat{S}, \hat{Q}) \in (P, S, Q)$, i.e. \hat{S} is a subprogram of S with local specification (\hat{P}, \hat{Q}) . If $\models \text{VCG}^w(P, S, Q)$ then $\models \text{VCG}^w(\hat{P}, \hat{S}, \hat{Q})$.*

Proof. By induction on the definition of the \subseteq relation. The first two cases are trivial. If $S = C_1; \dots; C_n$ and $C_i = \mathbf{if } b \mathbf{ then } S_t \mathbf{ else } S_f$ for some $i \in \{1, \dots, n\}$, Lemma 2 (2) yields $\models \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge b, S_t, \overline{\text{wprec}}_{i+1}(S, Q))$ and $\models \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{i+1}(S, Q))$, as desired. Finally, if $S = C_1; \dots; C_n$ and $C_i = \mathbf{while } b \mathbf{ do } \{I\} S_b$ for some $i \in \{1, \dots, n\}$, Lemma 2 (3) yields $\models \text{VCG}^w(I \wedge b, S_b, I)$. \square

Definition 75 (Labeled Control Flow Graph). *Given a program S , precondition P and postcondition Q such that $S = C_1; \dots; C_n$, the **labeled control flow graph** $\text{LCFG}(S, P, Q)$ of S with respect to (P, Q) is a labeled directed acyclic graph (DAG) whose edge labels are pairs of logical assertions on program states. To each command C in the program S we associate an input node $\text{IN}(C)$ and an output node $\text{OUT}(C)$. The graph is constructed as follows:*

1. *Each command C_i in S will be represented by one (in the case of **skip** and assignment commands) or two nodes (for conditional and loop commands):*

- *If C_i is **skip** or an assignment command, there is a new node C_i in the graph.
We set $\text{IN}(C_i) = \text{OUT}(C_i) = C_i$.*
- *If $C_i = \mathbf{if } b \mathbf{ then } S_t \mathbf{ else } S_f$, there is two new nodes $\text{if}(b)$ and fi in the graph.
We set $\text{IN}(C_i) = \text{if}(b)$ and $\text{OUT}(C_i) = \text{fi}$.*
- *If $C_i = \mathbf{while } b \mathbf{ do } \{I\} S$ or $C_i = \mathbf{while } b \mathbf{ do } \{I, e_v\} S$, there is two new nodes $\text{do}(b)$ and od in the graph.
We set $\text{IN}(C_i) = \text{do}(b)$ and $\text{OUT}(C_i) = \text{od}$.*

2. *Let $\text{LCFG}(S, P, Q)$ also contain two additional nodes START and END .*

3. *Let $\text{LCFG}(S, P, Q)$ contain an edge $(\text{OUT}(C_i), \text{IN}(C_{i+1}))$ for $i \in \{1, \dots, n-1\}$, and two additional edges $(\text{START}, \text{IN}(C_1))$ and $(\text{OUT}(C_n), \text{END})$.*

The labels of these edges are set as follows:

$$\begin{aligned} \text{lb}(\text{START}, \text{IN}(C_1)) &= (\overline{\text{spost}}_0(S, P), \overline{\text{wprec}}_1(S, Q)) \\ &= (P, \overline{\text{wprec}}_1(S, Q)); \end{aligned}$$

$$\text{lb}(\text{OUT}(C_i), \text{IN}(C_{i+1})) = (\overline{\text{spost}}_i(S, P), \overline{\text{wprec}}_{i+1}(S, Q));$$

$$\text{lb}(\text{OUT}(C_n), \text{END}) = (\overline{\text{spost}}_n(S, P), \overline{\text{wprec}}_{n+1}(S, Q)) = (\overline{\text{spost}}_n(S, P), Q).$$

4. *For $i \in \{1, \dots, n\}$, if $C_i = \mathbf{if } b \mathbf{ then } S_t \mathbf{ else } S_f$, we recursively construct the graphs*

$$\text{LCFG}(S_t, b \wedge \overline{\text{spost}}_{i-1}(S, P), \overline{\text{wprec}}_{i+1}(S, Q))$$

and

$$LCFG(S_f, \neg b \wedge \overline{\text{spost}}_{i-1}(S, P), \overline{\text{wprec}}_{i+1}(S, Q))$$

These graphs are grafted into the present graph by removing their *START* nodes and setting the origin of the dangling edges to be in both cases the node $IN(C_i)$, and similarly removing their *END* nodes and setting the destination of the dangling edges to be the node $OUT(C_i)$.

5. For $i \in \{1, \dots, n\}$, if $C_i = \mathbf{while} \ b \ \mathbf{do} \ \{I\} \ S$, we recursively construct the graph

$$LCFG(S, I \wedge b, I)$$

or

$$LCFG(S, I \wedge b \wedge e_v = x_0, I \wedge e_v < x_0)$$

if a loop variant is present, i.e. $C_i = \mathbf{while} \ b \ \mathbf{do} \ \{I, e_v\} \ S$.

This graph is grafted into the present graph by removing its *START* node and setting the origin of the dangling edge to be the node $IN(C_i)$, and similarly removing its *END* node and setting the destination of the dangling edge to be the node $OUT(C_i)$.

Clearly every subprogram \hat{S} of S is represented by a subgraph of $LCFG(S, P, Q)$ delimited by a pair of nodes *START/END*, *if/fi*, or *do/od*. Let us denote these nodes respectively by $IN(\hat{S})$ and $OUT(\hat{S})$. The basic intuition of labeled CFGs is that for every pair of consecutive commands \hat{C}_i, \hat{C}_{i+1} in \hat{S} , there exists an edge $(\hat{C}_i, \hat{C}_{i+1})$ in $LCFG(S, P, Q)$ whose label consists of the strongest postcondition of the prefix of \hat{S} ending with \hat{C}_i , and the weakest precondition of the suffix of \hat{S} beginning with \hat{C}_{i+1} , with respect to the local specification (\hat{P}, \hat{Q}) of \hat{S} propagated from (P, Q) .

If $\models \mathbf{VCG}^w(\hat{P}, \hat{S}, \hat{Q})$ then every edge in the subgraph representing \hat{S} has a label (ϕ, ψ) such that $\models \phi \rightarrow \psi$, as a consequence of Lemma 2 (1). Moreover, by Lemma 4, if $\models \mathbf{VCG}^w(P, S, Q)$ then this must be true for every subprogram \hat{S} of S , and thus every edge in the graph $LCFG(S, P, Q)$ has a label (ϕ, ψ) such that $\models \phi \rightarrow \psi$.

If loops are annotated with variants, this is taken into account when constructing the subgraph corresponding to the loop's body (point 5 of the definition). So we now have that $\models \mathbf{VCG}_t^w(P, S, Q)$ implies $\models \phi \rightarrow \psi$ for every label (ϕ, ψ) in the graph.

To illustrate this concept, Figure 6.3 shows the LCFG for program 6.2 with respect to the specification $(y > 10, x \geq 0)$, where we simplify the strongest postconditions calculation for the sake of readability.

```

1  if (y > 0) then
   x := 100;
3  x := x+50;
   x := x-100
5  else
   x := x-150;
7  x := x-100;
   x := x+100

```

Listing 6.2: Example for LCFG

6.3 Verification Graphs

In this section the notion of Verification Graph and its properties are studied. In particular, it is shown how the LCFG previously introduced can be used for interactive verification.

Definition 76 (Verification Graph and Edge Conditions). *Let Π be a program and $\mathbf{p} \in \mathbf{PN}(\Pi)$ a procedure of Π . Then the verification graph of \mathbf{p} , $VC(\mathbf{p})$, is the graph $LCFG(\mathbf{body}(\mathbf{p}), \mathbf{pre}(\mathbf{p}), \mathbf{post}(\mathbf{p}))$. A formula $\phi \rightarrow \psi$ such that (ϕ, ψ) is the label of an edge in the verification graph of \mathbf{p} will be called an **edge condition** of \mathbf{p} . $EC(\mathbf{p})$ will denote the set of Edge Conditions of procedure \mathbf{p} .*

Naturally, we may also speak of the set of verification graphs of a program Π , which is the set of verification graphs of all its procedures.

After defining the concept of Verification Graph, it is possible to prove the following lemmas:

Lemma 5. *Let S be a block of commands and P, Q assertions.*

Then $VCG^w(P, S, Q) \subseteq EC(LCFG(S, P, Q))$.

Proof. We prove that

(i) $P \rightarrow \mathbf{wprec}(S, Q) \in EC(LCFG(S, P, Q))$ and

(ii) $VC^w(S, Q) \subseteq EC(LCFG(S, P, Q))$.

(i) is a straightforward consequence of the construction of LCFGs in Definition 75, point 3, since the label of the outgoing edge from node $START$ is $(P, \mathbf{wprec}(S, Q))$. (ii) is proved by induction on the structure of S . **skip**, assignment, and procedure call are trivial cases.

If S is **if** b **then** S_t **else** S_f , the graph will include (according to point 4 of the definition) the subgraphs $LCFG(S_t, b \wedge P, Q)$ and $LCFG(S_f, \neg b \wedge P, Q)$, and thus by induction hypothesis $VC^w(S_t, Q) \subseteq EC(LCFG(S, P, Q))$ and $VC^w(S_f, Q) \subseteq EC(LCFG(S, P, Q))$ as desired.

If S is **while** b **do** $\{I\}$ S_b , the graph will include (according to point 5 of the definition) the subgraph $LCFG(S_b, I \wedge b, I)$, and thus by induction hypothesis $VC^w(S_b, I) \subseteq EC(LCFG(S, P, Q))$. Note also that according to point 3 the outgoing edge from the node $OUT(S)$ (C^{od}) has label

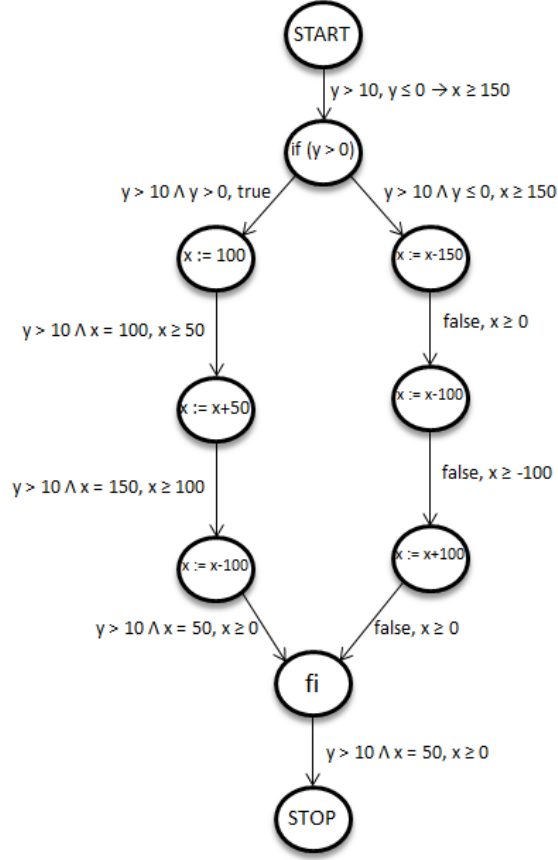


Figure 6.3: LCFG for program 6.2 with respect to the specification $(y > 10, x \geq 0)$

$lb((\text{spost}(S, P), Q), =)(I \wedge \neg b, Q)$, and again by point 3 the first edge in the graph $LCFG(S_b, I \wedge b, I)$ will have the label $(I \wedge b, \text{wprec}(S_b, I))$. Thus $\text{VC}^w(S, Q) \subseteq \text{EC}(LCFG(S, P, Q))$.

Finally, if the block S is a sequence of (more than one) commands $C; S'$, the graph $LCFG(S, P, Q)$ clearly has as subgraphs $LCFG(C, P, \text{wprec}(S', Q))$ and $LCFG(S', \text{wprec}(S', Q), Q)$ (except for the absence of the END node of the former and the $START$ node of the latter), from which it follows by induction hypothesis that $\text{VC}^w(C, \text{wprec}(S', Q)) \subseteq \text{EC}(LCFG(S, P, Q))$ and $\text{VC}^w(S', Q) \subseteq \text{EC}(LCFG(S, P, Q))$ as desired. \square

Lemma 6. *Given a procedure \mathbf{p} , let G be its verification graph, and moreover let $S = C_1; \dots; C_{i-1}; C_i; C_{i+1}; \dots; C_n$ be a block of commands of \mathbf{p} with local specification (P, Q) , i.e. $(P, S, Q) \in (\text{pre}(\mathbf{p}), \text{body}(\mathbf{p}), \text{post}(\mathbf{p}))$. Then*

- the label of the incoming edge into the node $IN(C_i)$ is

$$(\overline{\text{spost}}_{i-1}(S, P), \overline{\text{wprec}}_i(S, Q));$$

- the label of the outgoing edge from the node $OUT(C_i)$ is $(\overline{\text{spost}}_i(S, P), \overline{\text{wprec}}_{i+1}(S, Q))$.

Proof. Straightforward consequence of the construction of the graph in Definition 75. First note that graphs corresponding to subblocks of $\mathbf{body}(\mathbf{p})$ are constructed (points 4 and 5 of the definition) based on their local specifications, following Definition 74. Thus S will be represented by a subgraph $LCFG(S, P, Q)$ of G . The lemma then follows from point 3 of Definition 75. \square

Lemma 5 implies that the verification graph of a procedure contains a set of verification conditions for it (generated using weakest preconditions exclusively). In fact all the other edge conditions are valid when this set is valid:

Proposition 1. *For any procedure \mathbf{p} ,*

$$\models \text{VCG}^w(\mathbf{pre}(\mathbf{p}), \mathbf{body}(\mathbf{p}), \mathbf{post}(\mathbf{p})) \text{ iff } \models \text{EC}(\mathbf{p})$$

Proof. By Lemma 5 one has that $\text{VCG}^w(\mathbf{pre}(\mathbf{p}), \mathbf{body}(\mathbf{p}), \mathbf{post}(\mathbf{p})) \subseteq \text{EC}(\mathbf{p})$, which proves the *if* part. For the reverse implication, first note that by Lemma 4 every block of commands of \mathbf{p} has valid VCs with respect to its local specification, and thus as a consequence of Lemmas 6 and 2, for every command C in the body of \mathbf{p} one has that the labels of the incoming edge into the node $IN(C)$ and of the outgoing edge from the node $OUT(C)$ are pairs of assertions (ϕ, ψ) such that $\models \phi \rightarrow \psi$. \square

What is obtained is a control flow graph annotated with assertions (the edge conditions) from which different sets can be picked whose validity is sufficient to guarantee the correctness of the procedure. Each particular set corresponds to one particular verification strategy, mixing the use of strongest postconditions and weakest preconditions. The reason why there are different choices is that many edge conditions in the same graph are equivalent. Let us now try to characterize in precise terms these equivalences.

Every block of code in a procedure is represented as a set of paths in its verification graph (a single path if the block contains no branching), and the label of each edge in the path consists of the strongest postcondition of a prefix and the weakest precondition of a suffix of the block, with respect to its local specification. Now it is easy to see that in the case of atomic commands, the labels of adjacent edges correspond to equivalent assertions, and in the case of conditional commands it is the conjunction of labels of the branching edges that is equivalent to the adjacent edge.

Proposition 2. *In the conditions of Lemma 6, let (p_i, q_i) be the label of the incoming edge into node $IN(C_i)$ and (p_o, q_o) the label of the outgoing edge from node $OUT(C_i)$ in G . Then the following hold:*

1. *If C_i is of the form **skip** or $x := e$ or **call q**, then*

$$\models p_i \rightarrow q_i \text{ iff } \models p_o \rightarrow q_o$$

(note that $IN(C_i)$ and $OUT(C_i)$ are here the same node).

2. *If C_i is of the form **if b then S_t else S_f** , let (p_i^t, q_i^t) , (p_i^f, q_i^f) be the labels of the outgoing edges from $IN(C_i)$ and (p_o^t, q_o^t) , (p_o^f, q_o^f) the labels of the incoming edges into $OUT(C_i)$, corresponding to the then and else branch respectively. Then*

$$\begin{aligned} \models p_i \rightarrow q_i &\text{ iff } \models p_i^t \rightarrow q_i^t \text{ and } \models p_i^f \rightarrow q_i^f \\ \models p_o \rightarrow q_o &\text{ iff } \models p_o^t \rightarrow q_o^t \text{ and } \models p_o^f \rightarrow q_o^f \end{aligned}$$

Proof. Let $U = \overline{\text{spost}}_{i-1}(S, P)$ and $V = \overline{\text{wprec}}_{i+1}(S, Q)$. Following Lemma 6 and the definitions of weakest precondition and strongest postcondition, we have

$$\begin{aligned} p_i \rightarrow q_i &\equiv U \rightarrow \text{wprec}(C_i, V) \\ p_o \rightarrow q_o &\equiv \text{spost}(C_i, U) \rightarrow V \end{aligned}$$

1. $U \rightarrow \text{wprec}(C_i, V) \equiv \text{spost}(C_i, U) \rightarrow V$ holds when C_i is **skip**, $x := e$, or **call q**.
2. By definition of weakest precondition/ strongest postcondition and propositional equivalences, we have

$$\begin{aligned} p_i \rightarrow q_i &\equiv (b \wedge U \rightarrow \text{wprec}(S_t, V)) \wedge (\neg b \wedge U \rightarrow \text{wprec}(S_f, V)) \\ p_o \rightarrow q_o &\equiv (\text{spost}(S_t, b \wedge U) \rightarrow V) \wedge (\text{spost}(S_f, \neg b \wedge U) \rightarrow V) \end{aligned}$$

The labels of the branching edges are set by Definition 75.

The subgraphs $LCFG(S_t, b \wedge \overline{\text{spost}}_{i-1}(S, P), \overline{\text{wprec}}_{i+1}(S, Q))$ and $LCFG(S_f, \neg b \wedge \overline{\text{spost}}_{i-1}(S, P), \overline{\text{wprec}}_{i+1}(S, Q))$ are constructed in point 4, and by point 3 one then has

$$\begin{aligned} p_i^t \rightarrow q_i^t &\equiv b \wedge U \rightarrow \text{wprec}(S_t, V) \\ p_i^f \rightarrow q_i^f &\equiv \neg b \wedge U \rightarrow \text{wprec}(S_f, V) \\ p_o^t \rightarrow q_o^t &\equiv \text{spost}(S_t, b \wedge U) \rightarrow V \\ p_o^f \rightarrow q_o^f &\equiv \text{spost}(S_f, \neg b \wedge U) \rightarrow V \end{aligned}$$

□

This proposition implies that in a block not containing loops and conditionals, it is equivalent to check the validity of any edge condition in the block's path in the verification graph. If the block contains conditionals, it is indifferent to verify (i) an edge condition in the path before the conditional, or (ii) two edge conditions, one for each branch path, or (iii) an edge condition in the path after the conditional.

One way to formalize this is to assign a status to each edge. Colors **green** and **black** will be used for this. The idea is that the edges whose conditions are known to be valid (because they have been checked with a prover) are set to green, and we let this “checked” status propagate along the edges of the verification graph. If all edges become green then the procedure has been successfully verified.

Definition 77 (Edge Color in a Verification Graph). *Given a procedure \mathbf{p} let E be the set of edges of its verification graph and consider a subset $\mathcal{A} \subseteq \text{EC}(\mathbf{p})$ of its edge conditions. The function $\text{color}_{\mathcal{A}} : E \rightarrow \{\text{black}, \text{green}\}$ is defined as follows, where we write*

$$O \xrightarrow{(\phi, \psi)} D$$

for the edge with source O , destination D , and label (ϕ, ψ) (the label may be omitted when not relevant).

- If $\phi \rightarrow \psi \in \mathcal{A}$, then $\text{colour}_{\mathcal{A}}(O \xrightarrow{(\phi, \psi)} D) = \text{green}$
- If O corresponds to an atomic command and $\text{colour}_{\mathcal{A}}(N \rightarrow O) = \text{green}$ for some node N , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$
- If D corresponds to an atomic command and $\text{colour}_{\mathcal{A}}(D \rightarrow N) = \text{green}$ for some node N , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$
- If O is an “if” node and $\text{colour}_{\mathcal{A}}(N \rightarrow O) = \text{green}$ for some node N , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$ (note there are two such D for each node)
- If D is an “if” node and $\text{colour}_{\mathcal{A}}(D \rightarrow N_1) = \text{green}$ and $\text{colour}_{\mathcal{A}}(D \rightarrow N_2) = \text{green}$ for some nodes N_1, N_2 , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$
- If O is an “fi” node and $\text{colour}_{\mathcal{A}}(N_1 \rightarrow O) = \text{green}$ and $\text{colour}_{\mathcal{A}}(N_2 \rightarrow O) = \text{green}$ for some nodes N_1, N_2 , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$
- If D is an “fi” node and $\text{colour}_{\mathcal{A}}(D \rightarrow N) = \text{green}$ for some node N , then $\text{colour}_{\mathcal{A}}(O \rightarrow D) = \text{green}$ (note there are two such O for each node)

- Otherwise $\text{colour}_A(O \rightarrow D) = \text{black}$

The green color propagates freely through atomic command nodes in either direction. In branching nodes, it propagates from outside the conditional command into both branches; in the reverse direction, it will only propagate outwards when both branch edges are green.

Proposition 3. *Let \mathbf{p} be a procedure, E the set of edges of its verification graph, and $\mathcal{A} \subseteq \text{EC}(\mathbf{p})$ such that $\models \mathcal{A}$. If $\text{colour}_A(e) = \text{green}$ for every $e \in E$ then $\models \text{VCG}^w(\text{pre}(\mathbf{p}), \text{body}(\mathbf{p}), \text{post}(\mathbf{p}))$*

Proof. Every green edge has a valid associated condition. This can be shown by induction on the cardinality of the set of green edges, since the associated condition of every edge added to this set is either valid by hypothesis or equivalent to the condition of some edge already in the set (according to Proposition 2). The result then follows from Proposition 1. \square

A third color red could be additionally considered, for edges whose conditions have been proved to be invalid. For interactive verification, this would have the advantage of identifying the segments of the code where problems exist. red would also propagate freely across atomic command nodes; it would not propagate into conditional branches (since only one branch is known to have an invalid VC), but it would propagate outwards from any conditional branch (without requiring the other branch to also be red). In Chapter 9 it will be explained how these ideas have been incorporated in an interactive tool for program verification.

While loops have not yet been discussed. Each loop is represented by a pair of nodes *do*, *od*, and a path or set of paths from the former to the latter, corresponding to the loop's body. In *do* and *od* nodes there exists no equivalence between the edge conditions of the incoming and outgoing edges. Consider the subgraph

$$A \xrightarrow{(\phi_1, \psi_1)} \text{do} \xrightarrow{(\phi_2, \psi_2)} \dots \xrightarrow{(\phi_3, \psi_3)} \text{od} \xrightarrow{(\phi_4, \psi_4)} B$$

Then $\phi_1 \rightarrow \psi_1$ is the loop initialization VC, $\phi_2 \rightarrow \psi_2, \dots, \phi_3 \rightarrow \psi_3$ are invariant preservation VCs, and $\phi_4 \rightarrow \psi_4$ is the VC ensuring that the post-condition is granted by the invariant. Unlike the case of atomic command or branching nodes, in the presence of *do* / *od* nodes it is necessary to establish independently the validity of these VCs. In terms of coloring, these nodes *block* the propagation of the green color.

A consequence of this is that in the presence of an arbitrary path

$$A \xrightarrow{(\phi_1, \psi_1)} B \rightarrow \dots \rightarrow C \xrightarrow{(\phi_2, \psi_2)} D$$

if the path contains no loop nodes, then $\phi_1 \rightarrow \psi_1 \equiv \phi_2 \rightarrow \psi_2$, otherwise the equivalence does not hold: each loop introduces the need to prove three independent VCs.

Example

As an example consider the following procedure **partstep**

```

pre   $p \leq j \wedge j < r \wedge p - 1 \leq i \wedge i < j$ 
post  $p \leq j \wedge j \leq r \wedge p - 1 \leq i \wedge i < j$ 
proc partstep =
  if  $A[j] < x + 1$  then
     $i := i + 1;$ 
     $tmp := A[i];$ 
     $A[i] := A[j];$ 
     $A[j] := tmp$ 
  else skip;
   $j := j + 1$ 

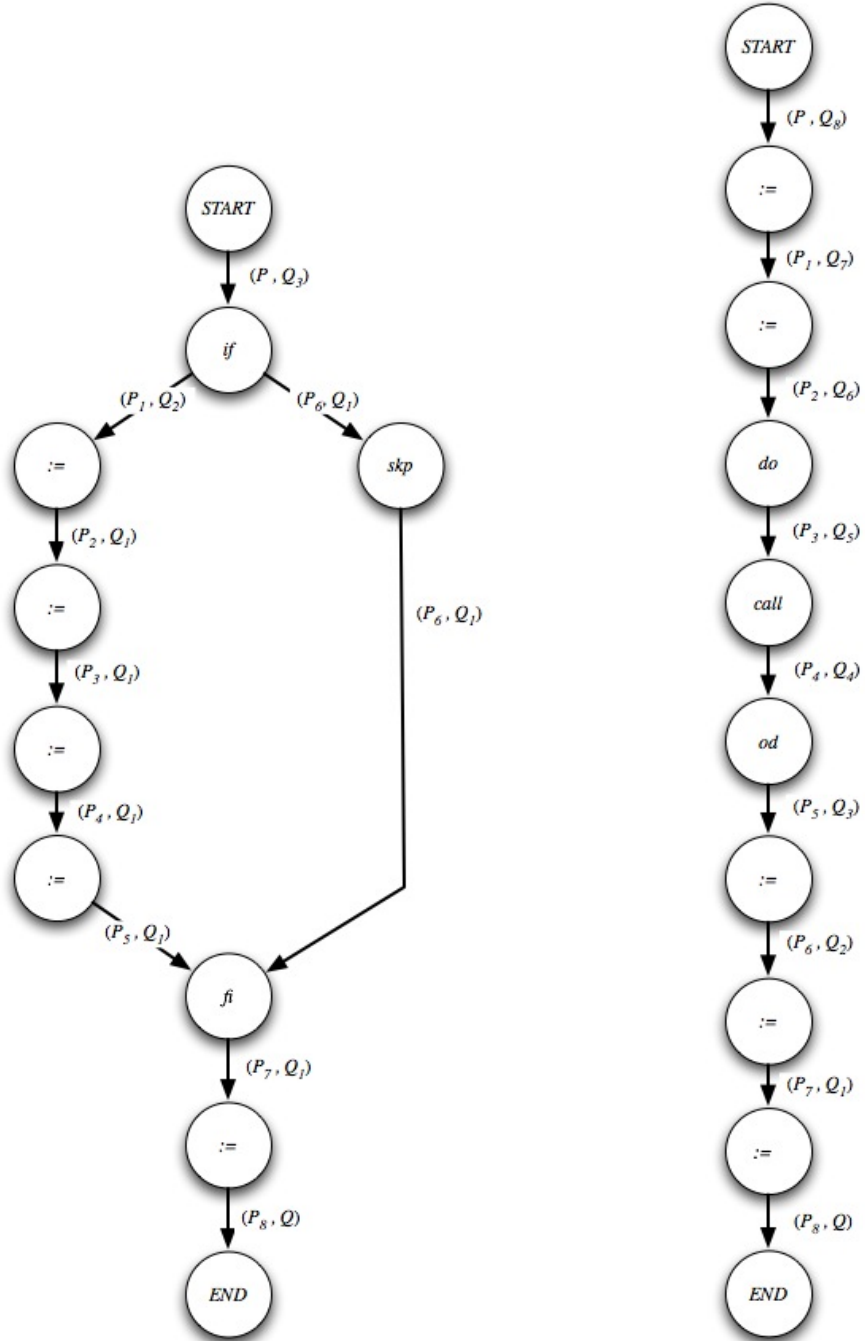
```

Its verification graph is shown in Figure 6.4, left, where:

$$\begin{aligned}
P &= p \leq j \wedge j < r \wedge p - 1 \leq i \wedge i < j \\
P_1 &\equiv A[j] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i \wedge i < j \\
P_2 &\equiv A[j] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j \\
P_3 &\equiv A[j] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j \wedge tmp = A[i] \\
P_4 &\equiv A[i] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j \wedge A[i] = A[j] \\
P_5 &\equiv A[i] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j \wedge A[j] = tmp \\
P_6 &\equiv \neg(A[j] < x + 1) \wedge p \leq j \wedge j < r \wedge p - 1 \leq i \wedge i < j \\
P_7 &\equiv (A[i] < x + 1 \wedge p \leq j \wedge j < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j \wedge A[j] = tmp) \\
&\quad \vee (\neg(A[j] < x + 1) \wedge p \leq j \wedge j < r \wedge p - 1 \leq i \wedge i < j) \\
P_8 &\equiv (A[i] < x + 1 \wedge p \leq j - 1 \wedge j - 1 < r \wedge p - 1 \leq i - 1 \wedge i - 1 < j - 1 \wedge A[j - 1] = tmp) \\
&\quad \vee (\neg(A[j - 1] < x + 1) \wedge p \leq j - 1 \wedge j - 1 < r \wedge p - 1 \leq i \wedge i < j - 1) \\
\\
Q &= p \leq j \wedge j \leq r \wedge p - 1 \leq i \wedge i < j \\
Q_1 &= p \leq j + 1 \wedge j + 1 \leq r \wedge p - 1 \leq i \wedge i < j + 1 \\
Q_2 &= p \leq j + 1 \wedge j + 1 \leq r \wedge p - 1 \leq i + 1 \wedge i + 1 < j + 1 \\
Q_3 &= (A[j] < x + 1 \rightarrow p \leq j + 1 \wedge j + 1 \leq r \wedge p - 1 \leq i + 1 \wedge i + 1 < j + 1) \\
&\quad \wedge (\neg(A[j] < x + 1) \rightarrow p \leq j + 1 \wedge j + 1 \leq r \wedge p - 1 \leq i \wedge i < j + 1)
\end{aligned}$$

Note that the contract of the procedure does not describe everything that the procedure does; in particular it only establishes the bounds of the variables j and i .

To prove that the local verification conditions of **p** are valid, one can choose to prove for instance any of $P \rightarrow Q_3$, or $P_8 \rightarrow Q$, or both $P_1 \rightarrow Q_2$ and $P_6 \rightarrow Q_1$. Now consider a second procedure that invokes **partstep** as follows.

Figure 6.4: Example verification graph: procedures **partstep** and **partition**

```

pre  $0 \leq p \wedge p \leq r$ 
post  $p \leq i + 1 \wedge i + 1 \leq r$ 
proc partition =
   $x := A[r];$ 
   $j := p;$ 
  while  $j < r$  do  $\{p \leq j \wedge j \leq r \wedge p - 1 \leq i \wedge i < j\}$ 
    call partstep
   $tmp := A[i + 1];$ 
   $A[i + 1] := A[r];$ 
   $A[r] := tmp;$ 

```

Its verification graph is shown in Figure 6.4, right. Establishing the conditional correctness of the program consisting of these two procedures would imply proving the following in addition to the previous condition:

- $P \rightarrow Q_8$ or $P_1 \rightarrow Q_7$ or $P_2 \rightarrow Q_6$, and
- $P_3 \rightarrow Q_5$ or $P_4 \rightarrow Q_4$, and
- $P_5 \rightarrow Q_3$ or $P_6 \rightarrow Q_2$ or $P_7 \rightarrow Q_1$, or $P_8 \rightarrow Q$.

Note in particular that $P_3 \rightarrow Q_5$ and $P_4 \rightarrow Q_4$ correspond to the preservation of the loop invariant, which implies reasoning with the contract of the **partstep** procedure (which in fact corresponds precisely to the preservation of the invariant).

In Chapter 9, it will be shown with concrete examples how this approach of verifying a program by employing verification graphs can be used to discover the incorrect statements in a program.

Chapter 7

Assertion-based Slicing

*I assert that nothing ever comes
to pass without a cause.*

Jonathan Edwards, 1703-1758

Program slicing [Wei81], as discussed in Chapter 4, is a well-established activity in software engineering. It plays an important role in program comprehension, since it allows software engineers to focus on the relevant portions of code (with respect to a given criterion). The basic idea is to isolate a subset of program statements that

- either directly or indirectly contribute to the values of a set of variables at a given program location, or
- are influenced by the values of a given set of variables.

Other statements are considered extraneous with respect to the given criterion and can be removed, enabling engineers to concentrate on the analysis of just the relevant ones. The first approach corresponds to *backward* forms of slicing, whereas the second corresponds to *forward* slicing.

Work in this area has focused on the development of progressively more effective, useful, and powerful slicing techniques, and has led to the use of these techniques in many application areas including program debugging, software maintenance, software reuse, and so on.

Program verification, as discussed in Chapter 3, on the other hand, is an apparently unrelated activity whose goal is to establish that a program performs according to some intended specification. Typically, what is meant by this is that the input/output behavior of the implementation matches that of the specification (this is usually called the *functional* behavior of the program), and moreover the program does not ‘go wrong’, for instance no errors occur during evaluation of expressions (the so-called *safety* behavior). As has been discussed in Chapter 6, modern program verification

systems are based on algorithms that examine a program and generate a set of *verification conditions* that are sent to an external theorem prover for checking. If all the conditions generated from a program can be proved, then the program is guaranteed to be correct with respect to the specification.

One point of contact that has been identified between slicing and verification is that traditional dependency-based slicing, applied a priori, facilitates the verification of large programs. This chapter explores the idea that it makes sense to slice programs based on semantic, rather than syntactic, criteria – the contracts used in DbC and program verification are excellent candidates for such criteria.

A typical example of a situation in which one could wish to calculate the slice of a program based on a specification is the reuse of annotated code. Suppose one is interested in reusing a module whose advertised contract consists of precondition P and postcondition Q , in situations in which a stronger precondition P' is known to hold, or else the desired postcondition Q' is weaker than the specified Q . Then from a software engineering perspective it would be desirable to eliminate, at source-level, the code that may be extraneous with respect to the specification (P', Q') .

From now on the expression “assertion-based slicing” will be used to refer to slicing methods based on the axiomatic semantics of programs, taking as criteria assertions (preconditions and/or postconditions) annotated in the programs. This includes *precondition-based* slicing, *postcondition-based* slicing, and *specification-based* slicing. Assertion-based slicing is more powerful and flexible than syntactic slicing, since the criteria can be as expressive as any set of first-order formulas on the initial and final states of the program. One of the first forms of slicing based on program semantics was *conditioned slicing* [CCL98], a form of forward slicing. This was shown to subsume both static and dynamic notions of dependency-based slicing, since the initial state of execution is constrained by a first-order formula that can be used to restrict the set of admissible initial states to exactly one (corresponding to dynamic slicing), or simply to identify a relevant subset of the state to be used as slicing criterion (as in static slicing). The same applies to backward slicing: using a postcondition as slicing criterion instead of a set of variables is clearly more expressive. Naturally, this expressiveness comes at a cost, since semantic forms of slicing are harder to compute.

Although the basic ideas have been published for over 10 years now, assertion-based slicing is still not very popular – in particular we are not aware of working tools that implement the ideas. The widespread usage of code annotations as explained above is however an additional argument for promoting it.

This chapter reviews (and clarifies aspects of) previous work in this area, sets a basis for slicing programs annotated with loop invariants and variants, and studies properties and applications of such slices. Moreover, new ideas are introduced, which allow us to develop an algorithm for specification-

based slicing that improves on previous algorithms in two aspects: the identification of sequences of statements that can be safely removed from a program (without modifying its semantics), and the selection of the biggest set of such sequences. Note that removable sequences may overlap, so this is not a trivial problem. This problem is solved by giving another application to the LCFG introduced in the previous chapter. While in Chapter 6 LCFGs were used as verification graphs, this Chapter introduces the notion of *slice graph*, which contains as subgraph the control flow graph of every slice of the program. This allow us to define a slicing algorithm that can be applied to calculate precondition-, postcondition-, and specification-based slices, but the focus will be on the latter, since the first two are particular cases.

One of the claims made in this chapter is that this new algorithm produces minimal slices. Note that the algorithm is *optimal in a relative sense*, since the test for removable subprograms involves first-order formulas whose validity must be established externally by some proof tool. Undecidability of first-order logic destroys any hope of being able to identify every removable subprogram automatically, since some valid formulas may not be proved.

Auxiliary Notation. Throughout the chapter, the following notation will be used to denote the sequence of commands S' obtained from S by removing a subsequence of commands. For $1 \leq i \leq j \leq n$,

$$\text{remove}(i, j, S) = \begin{cases} \text{skip} & \text{if } i = 1 \text{ and } j = n, \\ C_1; \dots; C_{i-1}; C_{j+1}; \dots; C_n & \text{otherwise.} \end{cases}$$

In this chapter we will often use the word “program” to refer to a block of commands, rather than in the sense of Chapter 3 (a set of mutually-recursive procedures).

Structure of the chapter. In Section 7.1 previous work in this area is reviewed, including a discussion of aspects of the existing algorithms regarding their precision and minimality of the calculated slices. Section 7.2 formalizes the notions introduced in the previous section. Sections 7.3 and 7.4 contain the main technical contributions: first, the properties of specification-based slicing are studied and a precise test for identifying removable blocks of code is proposed, as well as a principle for slicing subprograms of a program; later we introduce the setting for a graph-based algorithm that computes minimal slices of a program with respect to a given specification. Section 7.5 discusses in detail one particular application of assertion-based slicing: the elimination of redundant code. In Section 7.6, other forms of slicing that have some points in common with the ones presented are reviewed.

```

2 x := x + 100;
  x := x + 50;
  x := x - 100

```

Listing 7.1: Example for postcondition-based slicing (1)

7.1 Assertion-based Slicing

In this section the notions of slicing based on preconditions and postconditions are discussed, as well as algorithms for calculating them. Other related approaches are discussed in Section 7.6, in particular the notions of forward and backward *conditioned slice*.

As referred above, the expression *assertion-based slicing* will be used to encompass *postcondition-based*, *precondition-based*, and *specification-based* forms of slicing, which will be considered in turn in what follows. It is important to keep in mind the distinction between the definition of some form of slicing (which states when a program is a slice of another based on a given criterion), and algorithms for computing such slices. The fact that definitions and algorithms have often been introduced simultaneously in the same papers may cause some confusion between the two. Typically a definition admits more than one slice based on the same criterion, and an algorithm computes one particular slice in accordance with the definition.

This section is intended to introduce the reader to the key concepts of slicing based on assertions, but also to identify some limitations in the published work, which will be solved in the rest of the chapter.

7.1.1 Postcondition-based Slicing

The idea of slicing programs based on their specifications was introduced by Comuzzi et al [CH96] with the notion of *predicate slice* (*p-slice*), also known as postcondition-based slice. To understand the idea of p-slices, consider a program S and a given postcondition Q . It may well be the case that some of the commands in the program do not contribute to the truth of Q in the final state of the program, i.e. their presence is not required in order for the postcondition to hold. In this case, the commands may be removed. A crucial point here is that the considered set of executions of the program is restricted to those that will result in the postcondition being satisfied upon termination. In other words, not every initial state is admissible – only those for which the *weakest precondition* of the program with respect to Q holds.

Consider for instance program in Listing 7.1. The postcondition $Q = x \geq 0$ yields the weakest precondition $x \geq -50$. If the program is executed in a state in which this precondition holds and the commands in lines 2 and 3 are removed from it, the postcondition Q will still hold. To convince ourselves of this, it suffices to notice that after execution of the instruction

```

1 x := x-150;
  x := x+100;
3 x := x+100

```

Listing 7.2: Example for postcondition-based slicing (2)

in line 1 in a state in which the weakest precondition is true, the condition $x \geq 50$ will hold, which is in fact stronger than Q .

To be more systematic, for a program of the form $C_1; \dots; C_n$ with postcondition Q , if $\models \overline{\text{wprec}}_i(S, Q) \rightarrow \overline{\text{wprec}}_j(S, Q)$, with $i < j$, the sequence $C_i; \dots; C_{j-1}$ can be removed. In particular, if $\models \overline{\text{wprec}}_i(S, Q) \rightarrow Q$, the sequence $C_i; \dots; C_n$ can be removed. For the previous example one have

$$\begin{aligned}\overline{\text{wprec}}_3(S, Q) &= x \geq 100, \\ \overline{\text{wprec}}_2(S, Q) &= x \geq 50, \\ \overline{\text{wprec}}_1(S, Q) &= x \geq -50.\end{aligned}$$

Now observe that $\models \overline{\text{wprec}}_2(S, Q) \rightarrow Q$, which means that the instructions in lines 2 to 3 can in fact be removed: the postcondition Q will still hold for the sliced program when it is executed in a state satisfying $x \geq -50$.

P-slices are of course *not unique*. For instance since $\models \overline{\text{wprec}}_3(S, Q) \rightarrow Q$ as well, one could have chosen to remove only the instruction in line 3. Informally it can be said that given a set of slices of a program with respect to the same postcondition, the best slice is the one in which the highest number of instructions is removed.

It is also important to understand that not only suffixes of a sequence of commands may be removed. Consider the postcondition $Q = x \geq 0$ for program in Listing 7.2, which yields the following weakest preconditions

$$\begin{aligned}\overline{\text{wprec}}_3(S, Q) &= x \geq -100, \\ \overline{\text{wprec}}_2(S, Q) &= x \geq -200, \\ \overline{\text{wprec}}_1(S, Q) &= x \geq -50\end{aligned}$$

Note that although $\not\models \overline{\text{wprec}}_1(S, Q) \rightarrow Q$, the commands in lines 1 and 2 can be removed because $\models \overline{\text{wprec}}_1(S, Q) \rightarrow \overline{\text{wprec}}_3(S, Q)$. If the statement in line 3 is executed in a state in which $x \geq -50$ then the postcondition $x \geq 0$ will hold.

Please notice that in the limit, the set of executions that leads to the postcondition being satisfied may be empty (if the weakest precondition of the program is a contradiction, say $x < 0 \wedge x > 10$), in which case there exist no slices – the formal definition to be given below will clarify this point. Another extreme situation occurs when the postcondition is a valid assertion, say $x < 0 \vee x > -10$, in which case the entire program is seen as irrelevant, and admits as a slice the trivial program **skip**.

Calculating p-slices

It is easy to see how p-slices of a sequence of commands $S = C_1 ; \dots ; C_n$ can be computed with respect to a postcondition Q . The first step is of course to calculate the weakest preconditions $\overline{\text{wprec}}_i(S, Q)$, for $1 \leq i \leq n$, and to store this information, say in the abstract syntax tree of S .

The next step is to iterate the following basic procedure that attempts to remove the subsequence $C_i ; \dots ; C_{j-1}$, with $1 \leq i < j \leq n$:

- If $\models \overline{\text{wprec}}_i(S, Q) \rightarrow \overline{\text{wprec}}_j(S, Q)$ then slice S to $\text{remove}(i, j-1, S)$

This involves a trade-off between the number of proof obligations generated (each of which results in a call to the prover) and the potential number of lines that will be removed from the program. Suppose for instance one limit ourselves to removing suffixes of the initial program. The smallest such slice can be calculated with a linear number of calls to the prover (on the length of S), by fixing $j = n + 1$ (thus $\overline{\text{wprec}}_j(S, Q) = Q$). It suffices, in the second step above, to initialize $i = 1$, and then execute the following loop: the prover is invoked with the formula $\overline{\text{wprec}}_i(S, Q) \rightarrow Q$; if unsuccessful then i is incremented and a new iteration of the loop takes place; otherwise the algorithm stops. The resulting slice is $C_1 ; \dots ; C_{i-1}$.

Notice that this is of course a conservative approach: failure of the prover to establish the validity of the first-order formula $\overline{\text{wprec}}_i(S, Q) \rightarrow Q$ does not mean that the formula is not valid, but this uncertainty implies that removing the sequence $C_i ; \dots ; C_n$ might result in a program that is not a slice of S , so the algorithm proceeds to the next candidate suffix.

As previously said, it is now easy to understand that the same program may contain more than one removable subsequences, including prefixes, suffixes, and sequences that are neither prefixes nor suffixes. Moreover, these removable sequences may well overlap. Thus it is clear that no linear-time algorithm can possibly detect all removable sequences leading to the smallest slice.

The Original Quadratic Time Algorithm

The algorithm proposed by Comuzzi runs in *quadratic time* on the length of the sequence. The algorithm first tries to slice the entire program by removing its longest removable suffix, and then repeats this task, considering successively shorter prefixes of the resulting program, and removing their longest removable suffixes. Schematically:

```

for  $j = n + 1, n, \dots, 2$ 
  for  $i = 1, \dots, j - 1$ 
    if  $\text{valid}(\overline{\text{wprec}}_i(S, Q) \rightarrow \overline{\text{wprec}}_j(S, Q))$  then  $S \leftarrow \text{remove}(i, j - 1, S)$ 

```

For instance in a program with 999 statements the following pairs (i, j) would be considered in this order:

$(1, 1000), (2, 1000), \dots, (999, 1000), (1, 999), (2, 999), \dots, (998, 999), (1, 998), \dots$

This algorithm may fail to remove the longest sequence. Consider that

$$\models \overline{\text{wprec}}_1(S, Q) \rightarrow \overline{\text{wprec}}_{800}(S, Q)$$

and

$$\models \overline{\text{wprec}}_{700}(S, Q) \rightarrow \overline{\text{wprec}}_{900}(S, Q)$$

Two subsequences may be sliced off, consisting respectively of commands 1 to 799 and 700 to 899. The algorithm will consider (and remove) the shorter sequence first, and in doing so will eliminate the possibility of the longer sequence being considered, since line 800 will be removed (and it may happen that $\overline{\text{wprec}}_1(S, Q)$ is not stronger than any remaining $\overline{\text{wprec}}_k(S, Q)$). The resulting slice is thus *not minimal*.

An Improved Quadratic Algorithm

An alternative to Comuzzi's algorithm can be described as follows. Start with the entire program and consider in turn successively shorter sequences as candidates to be removed. Thus in the 999 statements program, sequences in the order $(1, 1000), (1, 999), (2, 1000), (1, 998), (2, 999), (3, 1000), (1, 997), \dots$ will be considered. This would certainly remove the longest removable sequence.

This algorithm is however not optimal either. Consider the case in which

$$\models \overline{\text{wprec}}_1(S, Q) \rightarrow \overline{\text{wprec}}_{400}(S, Q)$$

$$\models \overline{\text{wprec}}_{600}(S, Q) \rightarrow \overline{\text{wprec}}_{1000}(S, Q)$$

and

$$\models \overline{\text{wprec}}_{200}(S, Q) \rightarrow \overline{\text{wprec}}_{800}(S, Q)$$

. The longest sequence will be sliced off (600 program lines), but this will preclude the possibility of eliminating two shorter sequences that would together consist of 800 program lines: removing the larger contiguous sequence does not necessarily result in the smallest slice.

It should now be clear that considering all sequences in any given order cannot guarantee that the minimal slice is computed.

The same is true for precondition-based and specification-based slices, discussed below. In Section 7.4 it will be shown that this problem can in general be formulated as a graph problem, which is one of the contributions of this PhD work.

```

1 x := x+100;
  x := x-200;
3 x := x+200

```

Listing 7.3: Example for precondition-based slicing

7.1.2 Precondition-based Slicing

Chung and colleagues [CLYK01] later introduced *precondition-based slicing* as the dual notion of postcondition-based slicing. The idea is still to remove statements whose presence does not affect properties of the final state of a program. The difference is that the considered set of executions of the program is now restricted directly through a first-order condition on the initial state. Statements whose absence does not violate any property of the final state of any such execution can be removed. This is the same as saying that the assertion calculated as the strongest postcondition of the program (resulting from propagating forward the given precondition) is not weakened in the computed slice.

As an example of a precondition-based slice, consider now Listing 7.3, and the precondition $P = x \geq 0$. The effect of the first two instructions is to weaken the precondition. If these instructions are sliced off and the resulting program is executed in a state in which P holds, whatever postcondition held for the initial program will still hold for the sliced program.

To be systematic, for a program of the form $C_1; \dots; C_n$ with precondition P , if $\models \overline{\text{spost}}_i(S, P) \rightarrow \overline{\text{spost}}_j(S, P)$, with $i < j$, the sequence $C_{i+1}; \dots; C_j$ can be removed. In particular, if $\models P \rightarrow \overline{\text{spost}}_j(S, P)$, the sequence $C_1; \dots; C_j$ can be removed. For the previous example we have

$$\begin{aligned}
\overline{\text{spost}}_1(S, P) &= \exists v. v \geq 0 \wedge x = v + 100 && \equiv x \geq 100, \\
\overline{\text{spost}}_2(S, P) &= \exists v. v \geq 100 \wedge x = v - 200 && \equiv x \geq -100, \\
\overline{\text{spost}}_3(S, P) &= \exists v. v \geq -100 \wedge x = v + 200 && \equiv x \geq 100
\end{aligned}$$

Please notice that $\models P \rightarrow \overline{\text{spost}}_2(S, P)$, thus the first two commands can be sliced off. Similarly to postcondition-based slicing, this is not limited to removing prefixes (even though only prefixes are considered by the linear time algorithm proposed in [CLYK01]). In the same example program, since in fact

$$\models \overline{\text{spost}}_1(S, P) \rightarrow \overline{\text{spost}}_3(S, P)$$

it could be alternatively slice off lines 2 and 3 of the program, which shows that removable sequences may overlap.

As a final example, consider a program containing branching, Listing 7.4 (top). Again slicing the program involves computing its strongest postcondition with respect to a given precondition P . Both branches consist of sequences of commands; even if the conditional command itself cannot be

```

1  if (x >= 0) then
   x := x+100;
3  x := x-200;
   x := x+200
5  else
   x := x-150;
   x := x-100;
   x := x+100
9
11 -----
11 if (x >= 0) then
13 x := x+200
15 else
   skip

```

Listing 7.4: Example for precondition-based slicing

sliced off, it may well be the case that the branch subprograms can be sliced. To this effect, the precondition is strengthened with the boolean condition and its negation respectively, and slice each branch with respect to these strengthened preconditions. Let S_1 be $x := x+100; x := x-200; x := x+200$ and S_2 be $x := x-150; x := x-100; x := x+100$. S_1 will be sliced with respect to $P_1 = P \wedge x \geq 0$ and S_2 with respect to $P_2 = P \wedge x \not\geq 0$.

Now let P be $x \geq 0$. Then $P_1 \equiv x \geq 0$ and P_2 is a contradiction, which means that $\models P_2 \rightarrow \text{spost}(S_2, P_2)$. Consequently, S_2 will be sliced to **skip**. This makes sense, since the precondition eliminates the possibility of execution of the *else* branch of the conditional. On the other hand the *then* branch is just the previous example (Listing 7.3). Thus program on top of Listing 7.4 can be precondition-sliced with respect to $x \geq 0$ as shown at the bottom of this Listing.

7.1.3 Specification-based Slicing

A *specification-based slice* can be calculated when both a precondition P and a postcondition Q are given for a program S . The set of relevant executions is restricted to those for which Q holds upon termination when the program is executed in a state satisfying P . Programs resulting from S by removing a set of statements, and which are still correct regarding (P, Q) , are said to be specification-based slices of S with respect to (P, Q) .

The method proposed in [CLYK01] to compute such slices is based on a theorem proved by the authors, which states that the composition, in any order, of postcondition-based slicing (with respect to postcondition Q) and precondition-based slicing (with respect to precondition P) produces a specification-based slice with respect to (P, Q) . As an example recall the program in Listing 6.2, page 193, and here transcribed in Listing 7.5, and the specification $(y > 10, x \geq 0)$. Precondition-based slicing will slice both sequences inside the conditional by strengthening the precondition $y > 10$ with the condition $y > 0$ and its negation respectively. In the second case this yields a contradiction, which will result in the *else* branch sequence

```

1  if (y > 0) then
   x := 100;
3  x := x+50;
   x := x-100
5  else
   x := x-150;
   x := x-100;
   x := x+100
-----
11 if (y > 0) then
13 x := 100
15 else
    skip

```

Listing 7.5: Example for specification-based slicing

```

1 x := x*x;
  x := x+100;
3 x := x+50

```

Listing 7.6: Example for specification-based slicing

being completely sliced off. The *then* sequence branch is not affected. Postcondition-based slicing with respect to $x \geq 0$ will then produce the sliced program shown at the bottom of the listing.¹

Although this method does compute specification-based slices, it does not compute minimal slices, as can be seen by looking at program in Listing 7.6 with specification $(\top, x \geq 100)$. One have:

$$\begin{aligned}
\overline{\text{spost}}_0(S, P) &= \top \\
\overline{\text{spost}}_1(S, P) &= \exists v. x = v * v \\
\overline{\text{spost}}_2(S, P) &= \exists w. (\exists v. w = v * v) \wedge x = w + 100 && \equiv \exists v. x = v * v + 100 \\
\overline{\text{spost}}_3(S, P) &= \exists w. (\exists v. w = v * v + 100) \wedge x = w + 50 && \equiv \exists v. x = v * v + 150
\end{aligned}$$

and

$$\begin{aligned}
\overline{\text{wprec}}_4(S, Q) &= x \geq 100 = Q \\
\overline{\text{wprec}}_3(S, Q) &= x \geq 50 \\
\overline{\text{wprec}}_2(S, Q) &= x \geq -50 \\
\overline{\text{wprec}}_1(S, Q) &= \top
\end{aligned}$$

It is obvious that the postcondition is satisfied after execution of the instruction in line 2, which means that if line 3 is removed the sliced program will still be correct with respect to $(\top, x \geq 100)$. However, precondition-based and postcondition-based slicing both fail in removing this instruction, since no forward implications are valid among the $\overline{\text{spost}}_i(S, P)$ or the $\overline{\text{wprec}}_i(S, Q)$.

¹In fact [CLYK01] advocates replacing the entire conditional command by one of the branches when the other branch is sliced to **skip**, but it is debatable whether this transformation can still be considered as a form of slicing.

$$\begin{array}{c}
\hline
\text{skip} \preceq C_1; \dots; C_n \\
\hline
\\
\hline
C_1; \dots; C_{i-1}; C_{j+1}; \dots; C_n \preceq C_1; \dots; C_i; \dots; C_j; \dots; C_n \\
(1 < i \leq j \leq n \text{ or } 1 \leq i \leq j < n) \\
\hline
\\
\hline
C'_i \preceq C_i \qquad (i \leq i \leq n) \\
\hline
C_1; \dots; C'_i; \dots; C_n \preceq C_1; \dots; C_i; \dots; C_n \\
\hline
\\
\hline
S'_1 \preceq S_1 \quad S'_2 \preceq S_2 \\
\hline
\text{if } b \text{ then } S'_1 \text{ else } S'_2 \preceq \text{if } b \text{ then } S_1 \text{ else } S_2 \\
\hline
\\
\hline
S' \preceq S \\
\hline
\text{while } b \text{ do } \{I\} S' \preceq \text{while } b \text{ do } \{I\} S \\
\hline
\end{array}$$

Figure 7.1: Definition of relation “is portion of”

Composing precondition-based and postcondition-based slicing will of course not solve this fundamental flaw. In Section 7.3 it is shown that the precise identification of removable statements requires the *simultaneous* use of both preconditions and postconditions; trying to identify removable statements using only preconditions or only postconditions may fail.

7.2 Formalization of Assertion-based Slicing

In this section the notions of slicing discussed in the previous section are formalized. A program S' is a *specification-based slice* of S if it is a *portion* of S and moreover S can be *refined* to S' with respect to a given specification (a semantic notion). The notions of precondition-based and postcondition-based slice can be defined as special cases of this notion.

From now on, $S' \preceq S$ will be written with the meaning that program S' results from S by removing some statements. S' is said to be a *portion* or a *reduction* of S .

Definition 78 (Portion-of relation). *The $\cdot \preceq \cdot$ relation is the reflexive transitive closure of the relation generated by the set of axioms and rules given in Figure 7.1.*

Note that since Figure 7.1 defines an anti-symmetric relation, $\cdot \preceq \cdot$ is a

partial-order. As will be shortly seen, slices of a program S are portions of S that satisfy additional constraints.

Definition 79 (Assertion-based slices). *Let S be a program and (P, Q) a specification consisting of precondition P and postcondition Q . The program S' is said to be*

- a specification-based slice of S with respect to (P, Q) , written $S' \triangleleft_{(P,Q)} S$, if $S' \preceq S$ and

$$\models \text{VCG}^w(P, S, Q) \quad \text{implies} \quad \models \text{VCG}^w(P, S', Q)$$

- a precondition-based slice of S with respect to P , if $S' \triangleleft_{(P, \text{spost}(S, P))} S$;
- a postcondition-based slice of S with respect to postcondition Q if $S' \triangleleft_{(\text{wprec}(S, Q), Q)} S$.

Observe that it only makes sense to calculate specification-based slices of correct programs; if S is not correct with respect to (P, Q) then any portion of it is a slice with respect to (P, Q) . This does not however mean that techniques based on these forms of slicing cannot be applied to incorrect programs: they can be used on subprograms (proved correct) of incorrect programs. For instance, in Section 7.5 it will be discussed how postcondition-based slicing can be used for debugging purposes.

Notice also that the definitions of precondition-based and postcondition-based slicing are very strong, as the following lemma shows.

Lemma 7.

1. If S' is a precondition-based slice of S with respect to P , then for any assertion Q , $S' \triangleleft_{(P,Q)} S$
2. If S' is a postcondition-based slice of S with respect to Q , then for any assertion P , $S' \triangleleft_{(P,Q)} S$

Proof. We prove 1 (the proof of 2 is similar). We assume $\models \text{VCG}^w(P, S, Q)$, and thus by Lemma 1 $\models \text{spost}(S, P) \rightarrow Q$, $\text{VC}^s(S, P)$, and have to prove that $\models \text{VCG}^w(P, S', Q)$. Since S' is a precondition-based slice of S with respect to P , by Lemma 1 we have that

$$\begin{aligned} & \models \text{spost}(S, P) \rightarrow \text{spost}(S, P), \text{VC}^s(S, P) \\ & \quad \text{implies} \\ & \models \text{spost}(S', P) \rightarrow \text{spost}(S, P), \text{VC}^s(S', P) \end{aligned}$$

The left-hand side follows from our assumptions, and thus

$$\models \text{spost}(S', P) \rightarrow Q, \text{VC}^s(S', P)$$

which by the same lemma is equivalent to $\models \text{VCG}^w(P, S', Q)$. \square

Observe at this point that there are several differences between the definitions previously presented and those used in [CH96, CLYK01]. A first difference concerns all the above notions of slicing: previous notions require the weakest precondition (resp. strongest postcondition) to be *exactly the same* in the sliced program as in the original program, whereas now it allows to be *weaker* (resp. *stronger*), which is more coherent with the idea of the slice refining the behavior of the original program.

A second difference concerns specifically the definitions of precondition-based and postcondition-based slicing only. While the definitions given in [CLYK01] are based on implicative assertions relating the strongest postconditions (resp. weakest preconditions) of both programs, they are now explicitly defined as particular cases of specification-based slices, which is more convenient given our treatment of iteration through the use of annotated invariants. The following lemma makes the relation between both definitions explicit, for the case of programs without iteration.

Lemma 8. *Let S' be a program containing no loops. Then*

1. *If $S' \preceq S$ and $\models \text{spost}(S', P) \rightarrow \text{spost}(S, P)$, then $S' \triangleleft_{(P, \text{spost}(S, P))} S$.*
2. *if $S' \preceq S$ and $\models \text{wprec}(S, Q) \rightarrow \text{wprec}(S', Q)$, then $S' \triangleleft_{(\text{wprec}(S, Q), Q)} S$*

Proof.

1. Note that $\text{VCG}^s(P, S, \text{spost}(S, P)) = \text{spost}(S, P) \rightarrow \text{spost}(S, P)$, which is valid, and $\text{VCG}^s(P, S', \text{spost}(S, P)) = \text{spost}(S', P) \rightarrow \text{spost}(S, P)$. Thus $\models \text{VCG}^s(P, S, \text{spost}(S, P))$ implies $\models \text{VCG}^s(P, S', \text{spost}(S, P))$ and by Lemma 1 we have that $\models \text{VCG}^w(P, S, \text{spost}(S, P))$ implies $\models \text{VCG}^w(P, S', \text{spost}(S, P))$
2. Similar to 1.

□

The definitions above are formulated in a partial correctness setting, which means that terminating programs admit non-terminating programs as specification-based slices, and vice versa (it is easy to see that removing a single instruction from the body of a terminating loop may make it non-terminating, and vice versa). *Termination-sensitive* notions of slicing will now be introduced, by shifting from a partial correctness to a total correctness setting. A terminating program does not admit non-terminating programs as termination-sensitive slices.

Definition 80 (Termination-sensitive assertion-based slices). *Let S be a program and (P, Q) a specification consisting of precondition P and postcondition Q . The program S' is said to be*

- a termination-sensitive specification-based slice of S with respect to (P, Q) , written $S' \blacktriangleleft_{(P,Q)} S$, if $S' \preceq S$ and moreover

$$\models \text{VCG}_t^w(P, S, Q) \quad \text{implies} \quad \models \text{VCG}_t^w(P, S', Q)$$

- a termination-sensitive precondition-based slice of S with respect to P if $S' \blacktriangleleft_{(P, \text{spost}(S, P))} S$;
- a termination-sensitive postcondition-based slice of S with respect to Q if $S' \blacktriangleleft_{(\text{wprec}(S, Q), Q)} S$.

In the same way that, when verifying a program, one may proceed by first checking its partial correctness and then its termination to ensure total correctness, one may assert that S' is a termination-sensitive slice of the totally correct program S by checking that $S' \blacktriangleleft_{(P,Q)} S$ and additionally checking that S' terminates.

7.3 Properties of Assertion-based Slicing

In abstract terms, given a program $S = C_1; \dots; C_n$ with specification (P, Q) , an assertion-based slicing algorithm must be able to

1. Identify subprograms that *could* be removed from the program being sliced, while preserving its correctness with respect to a given specification. More concretely, the algorithm must decide for every i, j if

$$\text{remove}(i, j, S) \blacktriangleleft_{(P,Q)} S$$

holds, and then proceed recursively to identify removable subprograms of each C_i .

2. Select, among the set of statements identified as removable, the combination (or one of the combinations) that results in the best slice according to some criterion (the most obvious is the smallest number of program lines). Although this has been considered more seriously in the work of Comuzzi and colleagues on postcondition-based slicing, it applies to all three forms of slicing considered so far.

This section is devoted to point 1; the second point will be considered in Section 7.4.

The currently available algorithms for precondition-based and postcondition-based slicing check the validity of a formula relating the propagated conditions near the statements i and j . This seemed to be a good test of whether the sequence of commands between i and j could be removed, but in Section 7.1 it was shown that for precondition-based slicing the method used in previous work fails to identify statements that should be removed

because they do not contribute to the final state of the program, in any of the executions specified by the precondition. The failure occurs when the commands are made irrelevant by other instructions that occur *later* in the program, and thus cannot be detected by the prescribed method. The bottom line is that using Lemma 8 to design precondition or postcondition-based slicing algorithms is in fact misleading. The problem can be solved by simply observing our definition of these slices (Definition 79), which are given as particular cases of specification-based slices. For instance given a precondition P , it suffices to calculate the strongest postcondition of the program with respect to P and then calculate a specification-based slice of S with respect to $(P, \text{spost}(S, P))$.

For specification-based slicing, the algorithm of [CLYK01] considers sequentially the propagation of preconditions and postconditions. But in Section 7.1.3 it was shown that first slicing with preconditions and later with postconditions (or vice versa) may fail to remove statements which can be removed, according to the definition. It will be now shown that using preconditions and postconditions *simultaneously* allows for a precise identification of removable statements.

7.3.1 Removable Commands

Let us start by generalizing Lemma 1. This lemma states that there exist two equivalent ways to calculate verification conditions for a given program and specification: one based on weakest preconditions and another based on strongest postconditions. It will be now shown that for a given program this can be generalized: one can equally generate verification conditions by breaking the sequence of commands at any point, resulting in a prefix and a suffix of the initial command. The set of verification conditions is given as the union of the verification conditions of the suffix (computed using weakest preconditions) and of the prefix (using strongest postconditions). An additional verification condition relates the strongest postcondition of the prefix and the weakest precondition of the suffix.

The first point in the Lemma 2, page 186, formalizes this idea. Its significance is that, according to the following proposition, it can be decided when the sequence $C_i; \dots; C_j$ can be removed by considering the prefix $C_1; \dots; C_{i-1}$ and the suffix $C_{j+1}; \dots; C_n$.

Proposition 4. *Let (P, Q) be a specification, $S = C_1; \dots; C_n$ a program, and i, j , integers such that $1 \leq i \leq j \leq n$.*

If $\models \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q)$ then $\text{remove}(i, j, S) \triangleleft_{(P, Q)} S$

Proof. $\text{remove}(i, j, S)$ is clearly a portion of S . Now let us assume that $\models \text{VCG}^w(P, S, Q)$; we need to prove that $\models \text{VCG}^w(P, \text{remove}(i, j, S), Q)$.

Applying Lemma 2 (1) to S with $k = i - 1$ and $k = j$ we get respectively:

$$\begin{aligned} \models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_{i-1}^s(S, P), \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_i(S, Q), \\ & \overline{\text{VC}}_i^w(S, Q) \\ \models \text{VCG}^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_j^s(S, P), \overline{\text{spost}}_j(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q), \\ & \overline{\text{VC}}_{j+1}^w(S, Q) \end{aligned}$$

Thus $\models \overline{\text{VC}}_{i-1}^s(S, P)$ and $\models \overline{\text{VC}}_{j+1}^w(S, Q)$. Now it suffices to apply Lemma 2 (1) to the program $\text{remove}(i, j, S)$ with $k = i - 1$.

Since $\text{remove}(i, j, S) = C_1; \dots; C_{i-1}; C_{j+1}; \dots; C_n$, this yields the following, which we apply from right to left.

$$\begin{aligned} \models \text{VCG}^w(P, \text{remove}(i, j, S), Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_{i-1}^s(S, P), \\ & \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q), \\ & \overline{\text{VC}}_{j+1}^w(S, Q) \end{aligned}$$

□

Please notice that since

$$\models \text{VCG}^w(P, S, Q) \text{ implies } \models \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_i(S, Q)$$

and

$$\models \overline{\text{spost}}_j(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q)$$

the following also hold:

$$\text{If } \models \overline{\text{wprec}}_i(S, Q) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q) \text{ then } \text{remove}(i, j, S) \triangleleft_{(P, Q)} S \quad (7.1)$$

$$\text{If } \models \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{spost}}_j(S, P) \text{ then } \text{remove}(i, j, S) \triangleleft_{(P, Q)} S \quad (7.2)$$

However, the latter conditions are both stronger than the one in the proposition, which means that using them as tests could fail to identify some removable subprograms. This is in accordance with the examples in Section 7.1, which have shown that simply propagating P forward and Q backward, and checking for implications between the propagated $\overline{\text{spost}}_k(S, P)$ and then for implications between the propagated $\overline{\text{wprec}}_k(S, Q)$, while sound, might result in slices that are not minimal. The method proposed in the literature calculates slices using these stronger tests, and this is the reason why they fail, for instance in Listing 7.6. To illustrate our point with the latter program it suffices to notice that since $\models \overline{\text{spost}}_2(S, P) \rightarrow \overline{\text{wprec}}_4(S, Q)$, the command C_3 can be removed according to the test of Proposition 4.

Proposition 4 in fact provides us with the *weakest* condition for slicing programs, since the initial program is assumed to be correct with respect to the given specification the reverse implication also holds:

Proposition 5. *Let (P, Q) be a specification, $S = C_1 ; \dots ; C_n$ a program such that $\models \text{VCG}^w(P, S, Q)$, and i, j , integers such that $1 \leq i \leq j \leq n$.*

If $\text{remove}(i, j, S) \triangleleft_{(P, Q)} S$ then $\models \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q)$

Proof. It suffices to prove that $\text{VCG}^w(P, \text{remove}(i, j, S), Q)$ implies $\models \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q)$. Again applying Lemma 2 (1) to $\text{remove}(i, j, S)$ with $k = i - 1$ yields the following, which we now apply from left to right.

$$\begin{aligned} & \models \text{VCG}^w(P, \text{remove}(i, j, S), Q) \\ & \text{iff} \\ & \models \overline{\text{VC}}_{i-1}^s(S, P), \overline{\text{spost}}_{i-1}(S, P) \rightarrow \overline{\text{wprec}}_{j+1}(S, Q), \overline{\text{VC}}_{j+1}^w(S, Q) \end{aligned}$$

□

This is a guarantee that our test identifies all removable subsequences of commands of a correct program – note that implications (7.1) and (7.2) cannot be reversed in this way.

Finally, note that the results in this section apply equally in the scope of total correctness and termination-sensitive slicing (the proofs are similar, using Lemma 3 instead of Lemma 1). In particular, Lemma 2 (1) and (2) hold with VCG_t^w (resp. $\overline{\text{VC}}_t^w, \overline{\text{VC}}_t^s$) substituted for VCG^w (resp. $\overline{\text{VC}}^w, \overline{\text{VC}}^s$). (3) is stated as follows: if $C_k = \mathbf{while} \ b \ \mathbf{do} \ \{I, e_v\} S_b$ for some $k \in \{1, \dots, n\}$, then

$$\begin{aligned} \models \text{VCG}_t^w(P, S, Q) \quad \text{iff} \quad & \models \overline{\text{VC}}_t^s[k-1](S, P), \overline{\text{spost}}_{k-1}(S, P) \rightarrow I, \\ & I \wedge b \rightarrow e_v \geq 0, \\ & \text{VCG}_t^w(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0), \\ & I \wedge \neg b \rightarrow \overline{\text{wprec}}_{k+1}(S, Q), \overline{\text{VC}}_t^w[k+1](S, Q) \end{aligned}$$

Proposition 4 applies with $\cdot \blacktriangleleft_{(P, Q)} \cdot$ substituted for $\cdot \triangleleft_{(P, Q)} \cdot$ (removing any subsequence from a terminating sequence of commands cannot result in a non-terminating sequence, so this is not surprising). Proposition 5 also applies, with VCG_t^w substituted for VCG^w .

7.3.2 Slicing Subprograms

Let us now consider how the subprograms of S can be sliced with respect to a specification. Recall that conditional branching and loop commands are structurally composed of sequences of commands. These sequences (but not their subsequences) are called subprograms of the program under consideration. According to Definition 74 and the subsequent Lemma 4, given a specification (P, Q) for a program, to each of its subprograms is associated a local specification, which is obtained by propagating P and Q .

The following proposition states that slicing a subprogram of a program with respect to its local specification results in a slice of the program.

Proposition 6. *Let S, \hat{S} be programs such that $(\hat{P}, \hat{S}, \hat{Q}) \in (P, S, Q)$. Moreover let \hat{S}' be a portion of \hat{S} , i.e. $\hat{S}' \preceq \hat{S}$, and S' the program that results from replacing \hat{S} by \hat{S}' in S .*

1. *If $\hat{S}' \triangleleft_{(\hat{P}, \hat{Q})} \hat{S}$ then $S' \triangleleft_{(P, Q)} S$*
2. *If $\models \text{VCG}^w(P, S, Q)$ and $S' \triangleleft_{(P, Q)} S$ then $\hat{S}' \triangleleft_{(\hat{P}, \hat{Q})} \hat{S}$*

Proof. (1) Clearly S' must be a portion of S . The refinement aspect is proved by induction on the definition of \in as follows.

- If $\hat{S} = S$ the result holds trivially, with $S' = \hat{S}'$.
- Let $(P_1, S_1, Q_1) \in (P, S, Q)$ and $(\hat{P}, \hat{S}, \hat{Q}) \in (P_1, S_1, Q_1)$. Moreover let S'_1 be the program that results from replacing \hat{S} by \hat{S}' in S_1 . Then by induction hypothesis one has that $\hat{S}' \triangleleft_{(\hat{P}, \hat{Q})} \hat{S}$ implies $S'_1 \triangleleft_{(P_1, Q_1)} S_1$. Now let S' denote the result of replacing S_1 by S'_1 in S . Then again by induction hypothesis one has that $S'_1 \triangleleft_{(P_1, Q_1)} S_1$ implies $S' \triangleleft_{(P, Q)} S$. Note that S' can also be seen as the result of replacing \hat{S} by \hat{S}' in S , thus we are done since $\hat{S}' \triangleleft_{(\hat{P}, \hat{Q})} \hat{S}$ implies $S' \triangleleft_{(P, Q)} S$.
- Let $S = C_1; \dots; C_{i-1}; \text{if } b \text{ then } \hat{S} \text{ else } S_f; C_{i+1}; \dots; C_n$. We assume $\models \text{VCG}^w(P, S, Q)$; using Lemma 2 (2), this implies

$$\models \overline{\text{VC}}_{i-1}^s(S, P), \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge b, \hat{S}, \overline{\text{wprec}}_{i+1}(S, Q)), \\ \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{i+1}(S, Q)), \overline{\text{VC}}_{i+1}^w(S, Q)$$

and since $\hat{S}' \triangleleft_{(\overline{\text{spost}}_{i-1}(S, P) \wedge b, \overline{\text{wprec}}_{i+1}(S, Q))} \hat{S}$, this in turn implies

$$\models \overline{\text{VC}}_{i-1}^s(S, P), \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge b, \hat{S}', \overline{\text{wprec}}_{i+1}(S, Q)), \\ \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{i+1}(S, Q)), \overline{\text{VC}}_{i+1}^w(S, Q)$$

Now observe that since

$$S' = C_1; \dots; C_{i-1}; \text{if } b \text{ then } \hat{S}' \text{ else } S_f; C_{i+1}; \dots; C_n$$

again by Lemma 2 (2) one has $\models \text{VCG}^w(P, S', Q)$. The case when \hat{S} is the *else* branch is similar.

- Let $S = C_1; \dots; C_{i-1}; \text{while } b \text{ do } \{I\} \hat{S}; C_{i+1}; \dots; C_n$. We assume $\models \text{VCG}^w(P, S, Q)$; using Lemma 2 (3), this implies

$$\models \overline{\text{VC}}_{i-1}^s(S, P), \overline{\text{spost}}_{i-1}(S, P) \rightarrow I, \text{VCG}^w(I \wedge b, \hat{S}, I), \\ I \wedge \neg b \rightarrow \overline{\text{wprec}}_{i+1}(S, Q), \overline{\text{VC}}_{i+1}^w(S, Q)$$

and since $\hat{S}' \triangleleft_{(I \wedge b, I)} \hat{S}$, this in turn implies

$$\models \overline{\text{VC}}_{i-1}^s(S, P), \overline{\text{spost}}_{i-1}(S, P) \rightarrow I, \\ \text{VCG}^w(I \wedge b, \hat{S}', I), I \wedge \neg b \rightarrow \overline{\text{wprec}}_{i+1}(S, Q), \overline{\text{VC}}_{i+1}^w(S, Q)$$

Now observe that since

$$S' = C_1; \dots; C_{i-1}; \text{while } b \text{ do } \{I\} \hat{S}'; C_{i+1}; \dots; C_n$$

again by Lemma 2 (3) one has $\models \text{VCG}^w(P, S', Q)$.

(2) \hat{S}' is a portion of \hat{S} ; for the refinement aspect it suffices to prove that $\models \text{VCG}^w(P, S', Q)$ implies $\text{VCG}^w(\hat{P}, \hat{S}', \hat{Q})$. Again this is proved by induction on the definition of \subseteq . We illustrate this for the conditional case, with $S = C_1; \dots; C_{i-1}; \text{if } b \text{ then } \hat{S} \text{ else } S_f; C_{i+1}; \dots; C_n$. We have by Lemma 2 (2) that

$$\models \overline{\text{VC}}_{i-1}^s(S, P), \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge b, \hat{S}', \overline{\text{wprec}}_{i+1}(S, Q)), \\ \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge \neg b, S_f, \overline{\text{wprec}}_{i+1}(S, Q)), \overline{\text{VC}}_{i+1}^w(S, Q)$$

and thus $\models \text{VCG}^w(\overline{\text{spost}}_{i-1}(S, P) \wedge b, \hat{S}', \overline{\text{wprec}}_{i+1}(S, Q))$. \square

Recall that terminating programs admit non-terminating programs as specification-based slices, since termination-insensitive slicing merely forces the preservation of the loop invariants. In particular, since $\text{skip} \triangleleft_{(I \wedge b, I)} S$ always holds (because $\models I \wedge b \rightarrow I$), for any P, Q , and I one has that $\text{while } b \text{ do } \{I\} \text{ skip} \triangleleft_{(P, Q)} \text{while } b \text{ do } \{I\} S$. Consequently, any program admits as a slice the program that results from removing the body of every loop.

Of course, this does not apply in a total correctness setting. Again, if one substitutes $\cdot \triangleleft_{(P, Q)} \cdot$ for $\cdot \triangleleft_{(P, Q)} \cdot$ and VCG_t^w for VCG^w , the previous results are valid also in the context of termination-sensitive slicing (the proofs are similar). It suffices to consider an extra case in the definition of subprogram, for loops annotated with variants, as follows:

- If $S = C_1; \dots; C_n$ and $C_i = \text{while } b \text{ do } \{I, e_v\} S_b$ for some i with $1 \leq i \leq n$, then $(I \wedge b \wedge e_v = x_0, S_b, I \wedge e_v < x_0) \subseteq (P, S, Q)$.

The proposition can be used to further slice a program by slicing its subprograms with respect to their local specifications. Conditional branches are sliced by propagating the postcondition inside both branches, as well as the precondition strengthened with the boolean condition and its negation, respectively. In the case of a loop with invariant I and condition b , it suffices to use as specification for the body of the loop, the assertions $(I \wedge b, I)$, or $(I \wedge b \wedge e_v = x_0, I \wedge e_v < x_0)$ for termination-sensitive slicing.

This can be used in the following two scenarios. If the program is being sliced to remove redundant code, using a specification with respect to which it has been proved correct, then the loop annotations are adequate to prove correctness, and the proposition allows the removal of redundant code to proceed inside loops (each loop body can be sliced with respect to the preservation of its invariant and the strict decrease of its variant).

In a specialization / reuse scenario, the program is being sliced based on a weaker specification than the one with respect to which it was originally proved correct. In this scenario, it may well be the case that the loop invariants annotated in the program are stronger than they need to be – they have been used to establish correctness with respect to a stronger specification than the one now being used. In order to allow the slicing process to proceed usefully inside loop bodies, the user should first replace the invariants by weaker versions that are sufficient for proving correctness with respect to the new specification, and only then use Proposition 6 to slice the loop bodies with these weaker invariants.

Optimization Criterion. The above discussion raises a question concerning the criterion to be used for comparing the quality of different slices of the same program. The criterion that was implicit in Section 7.1 considered the total number of commands or lines of code in a slice. Note that for programs consisting only of atomic commands (i.e. for sequences of **skip** and assignment commands) the number of commands and lines (assuming one command per line) are the same. In the presence of commands containing subprograms however, this is not so, since our program syntax dictates that a loop or a conditional are a single command, regardless of the length of their body / branch subprograms. A more appropriate measure (i.e. closer to the notion of “lines of code”) is the number of *atomic commands and boolean conditions*.

Based on this criterion, selecting the minimal slice of a given program $S = C_1 ; \dots ; C_n$ implies taking into consideration the number of commands and boolean conditions of each subprogram \hat{S} of S – it makes no sense to just count the number (between 1 and n) of top-level commands in each slice of S . Moreover, since each \hat{S} can be sliced with respect to its local specification following Proposition 6, the general structure of a slicing algorithm should be to slice S only after slicing each of its subprograms with respect to its local specification. Only then can the minimal slice of S be selected.

7.3.3 Intermediate Conditions

A major difference between the notions of slicing based on assertions and traditional notions based on dependencies is that in the latter the slicing criterion always includes a *line number* k ; in forward slicing ask for instructions not dependent on the values at line k of a given set of variables to be

removed; in backward slicing it is the instructions on which the values of the variables at line k do not depend that are removed.

This can be mimicked in the current context by introducing an *intermediate assertion* to be taken into account for calculating a slice. Let us briefly explain how the framework under consideration can be extended in this direction.

Definition 81 (Specification-based Slice with Intermediate Condition). *Let $S = C_1 ; \dots ; C_n$ be a program, (P, Q) a specification, and R an assertion such that $\models \text{spost}(C_1 ; \dots ; C_k, P) \rightarrow R$ and $\models R \rightarrow \text{wprec}(C_{k+1} ; \dots ; C_n, Q)$. We say that the program $S' = S'_1 ; S'_2$ is a slice of S with respect to the specification (P, Q) and intermediate condition R at position k , written $S' \triangleleft_{(P,R,k,Q)} S$, if*

$$S'_1 \triangleleft_{(P,R)} C_1 ; \dots ; C_k \quad \text{and} \quad S'_2 \triangleleft_{(R,Q)} C_{k+1} ; \dots ; C_n$$

Although Definition 81 is sufficient to illustrate the idea, it can be generalized so that the intermediate condition regards some subprogram of S . Multiple intermediate conditions can also be admitted.

Naturally, such slices are particular cases of specification-based slices – the intermediate assertion simply restricts the definition further with respect to the specification. The following lemma is straightforward to prove using Lemma 2 (1).

Lemma 9. *If $S' \triangleleft_{(P,R,k,Q)} S$ then $S' \triangleleft_{(P,Q)} S$.*

Intermediate assertions can be used with the practical goal of facilitating automatic proofs by inserting conditions that are obviously true at the given line, which may allow more commands to be sliced off. But they also enrich the power of specification-based slicing, allowing one to slice fragments of the code with respect to local conditions, possibly even omitting part of the global specification. An extreme case is to compute a slice consisting of a postcondition-based slice of a prefix of a program, followed by a precondition-based slice of a suffix, as in

$$S' \triangleleft_{(\text{wprec}(C_1 ; \dots ; C_k, R), R, k, \text{spost}(C_{k+1} ; \dots ; C_n, R))} S.$$

In this case the intermediate condition is the only slicing criterion considered.

7.4 Slice Graphs

In the previous chapter, the application of labeled control flow graphs for the interactive generation of verification conditions was explored. In this chapter the same annotated graphs are used as the basis for the definition of slice graphs.

In a *slice graph*, the removable sequences of commands are associated with edges added to the initial control flow graph.

Definition 82 (Slice Graph). *Consider a program S and a specification (P, Q) such that $\models \text{VCG}^w(P, S, Q)$ (in which case we assume loops are not annotated with variants) or $\models \text{VCG}_t^w(P, S, Q)$. The slice graph $\text{SLCG}(S, P, Q)$ of S with respect to (P, Q) is obtained from the labeled control flow graph $\text{LCFG}(S, P, Q)$ by inserting additional edges as follows.*

For every subprogram $\hat{S} = \hat{C}_1; \dots; \hat{C}_n$, with $(\hat{P}, \hat{S}, \hat{Q}) \in (P, S, Q)$,

- *If $\models \hat{P} \rightarrow \hat{Q}$, a new **skip** node is inserted in the graph, together with two edges $(\text{IN}(\hat{S}), \text{skip})$ and $(\text{skip}, \text{OUT}(\hat{S}))$, both with label (\hat{P}, \hat{Q}) .*
- *For all $j \in \{1, \dots, n\}$ if $\models \hat{P} \rightarrow \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q})$, an edge $(\text{IN}(\hat{S}), \text{IN}(\hat{C}_{j+1}))$ with label $(\hat{P}, \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q}))$ is inserted;*
- *For all $i \in \{1, \dots, n\}$, if $\models \overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \hat{Q}$, an edge $(\text{OUT}(\hat{C}_{i-1}), \text{OUT}(\hat{S}))$ with label $(\overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}), \hat{Q})$ is inserted;*
- *For all $i, j \in \{1, \dots, n\}$ such that $i < j$, if $\models \overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q})$, an edge $(\text{OUT}(\hat{C}_{i-1}), \text{IN}(\hat{C}_{j+1}))$ with label $(\overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}), \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q}))$ is inserted;*

Note that this construction is purely based on the LCFG of S : \hat{P} , \hat{Q} , $\overline{\text{spost}}_{i-1}(\hat{S}, \hat{P})$, and $\overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q})$ can be read from labels of edges in the subgraph corresponding to the subprogram being considered. For each subprogram, first-order conditions are generated for every pair of edges such that the first precedes the second in the graph (the order in which this is done is irrelevant). If the validity of some condition cannot be established, the corresponding edge will not be added to the graph, in accordance with the requirement that slicing must be conservative.

As an example, Figure 7.2 partially shows the slice graph for program in Listing 6.2 with respect to the specification $(y > 10, x \geq 0)$. Removable sequences are signaled by the thick edges (and one **skip** node) that are added to the initial labeled CFG. Many edges are omitted to lighten the presentation of the graph; two edges are missing in the *then* branch (from $x := 100$ to $x := x - 100$ and from $x := x + 50$ to *fi*), and in the *else* branch five edges are missing – since the first component of every label in this path is a contradiction, an edge is inserted from each node to every reachable node in the *else* branch).

For any given subprogram \hat{S} of S , the slice graph contains as subgraph the LCFG of every slice of \hat{S} with respect to its local specification, and consequently also the LCFG of the program that results from replacing in

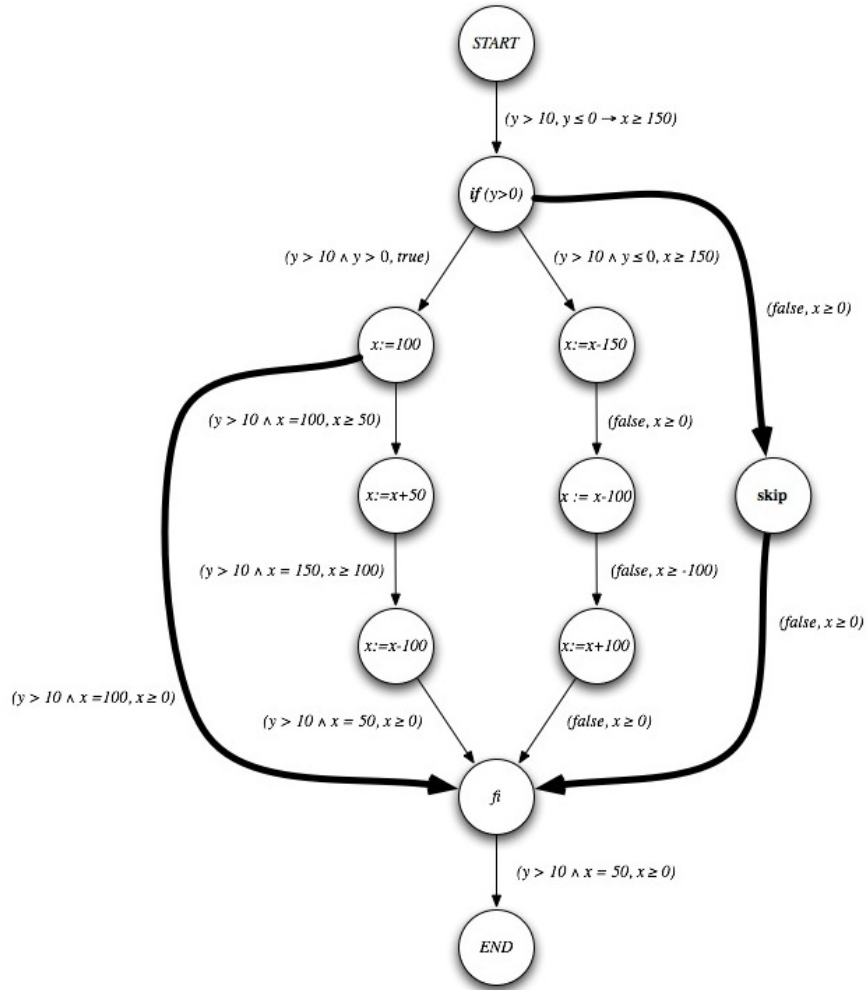


Figure 7.2: Example slice graph (extract). Thick lines represent edges that were added to the initial CFG, corresponding to “shortcut” subprograms that do not modify the semantics of the program. These paths have the same origin and destination nodes as other longer paths corresponding to removable sequences

S any subprogram by one of its slices. The following result formalizes this fact.

Lemma 10. *In the conditions of Definition 82, let i, j be integers such that $1 \leq i \leq j \leq n$, and S' the program resulting from replacing \hat{S} by $\text{remove}(i, j, \hat{S})$. Then*

1. *If $\models \text{VCG}^w(P, S, Q)$,*

$$\text{remove}(i, j, \hat{S}) \triangleleft_{(\hat{P}, \hat{Q})} \hat{S}$$

iff the graph $\text{LCFG}(S', P, Q)$ is a subgraph of $\text{SLCG}(S, P, Q)$.

2. *If $\models \text{VCG}_t^w(P, S, Q)$,*

$$\text{remove}(i, j, \hat{S}) \blacktriangleleft_{(\hat{P}, \hat{Q})} \hat{S}$$

iff the graph $\text{LCFG}(S', P, Q)$ is a subgraph of $\text{SLCG}(S, P, Q)$.

Proof. We prove 1 (the proof of 2 is similar, since propositions 4 and 5 also apply for termination-sensitive slicing, as explained at the end of Section 7.3.1).

(Only if part) By Proposition 5, $\models \overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q})$. Clearly the graph $\text{LCFG}(S', P, Q)$ is equal to $\text{LCFG}(S, P, Q)$, with the exception of a subgraph that is no longer present, and is replaced by an edge (or two edges and a **skip** node). Moreover these new edges are present in the graph $\text{SLCG}(S, P, Q)$, following Definition 82.

(If part) We prove that $\models \text{VCG}^w(\hat{P}, \text{remove}(i, j, \hat{S}), \hat{Q})$ (note that $\models \text{VCG}^w(\hat{P}, \hat{S}, \hat{Q})$ must hold, following Lemma 4). The graph $\text{LCFG}(S', P, Q)$ is the same as $\text{LCFG}(S, P, Q)$, except for the subgraphs corresponding to the commands removed inside \hat{S} .

If $\text{LCFG}(S', P, Q)$ is a subgraph of $\text{SLCG}(S, P, Q)$ then the edge or edges that short-circuit the subgraph corresponding to the removed commands can only have been introduced (by Definition 82) because $\models \overline{\text{spost}}_{i-1}(\hat{S}, \hat{P}) \rightarrow \overline{\text{wprec}}_{j+1}(\hat{S}, \hat{Q})$, and Proposition 4 allows us to conclude the proof. \square

A consequence of the previous result is that all slices of a program are represented in its slice graph, and no other portions of S are.

Proposition 7. *Let S, S' be programs such that $S' \preceq S$. Then*

1. *If $\models \text{VCG}^w(P, S, Q)$,*

$$S' \triangleleft_{(P, Q)} S$$

iff

the control flow graph $\text{LCFG}(S', P, Q)$ is a subgraph of $\text{SLCG}(S, P, Q)$.

2. If $\models \text{VCG}_t^w(P, S, Q)$,

$$S' \blacktriangleleft_{(P,Q)} S$$

iff

the control flow graph $\text{LCFG}(S', P, Q)$ is a subgraph of $\text{SLCG}(S, P, Q)$.

Proof. We prove 1 (the proof of 2 is similar, since Proposition 6 also applies for termination-sensitive slicing, as explained in Section 7.3.2).

(Only if part) The slice S' is a portion of S , which is obtained by removing some top-level commands of S and doing this recursively for some of the subprograms of S . By Proposition 6 (2), all the commands removed in every subprogram of S must result in slices regarding the local specification of the corresponding subprogram. Consider any sequencing of these subprogram slicing operations; the proof proceeds by induction on the length of this sequence using Lemma 10 (from left to right).

(If part) It is clear that in any control-flow graph that is a subgraph of $\text{SLCG}(S, P, Q)$, each edge (or pair of edges with a **skip** node) that is not present in the initial graph $\text{LCFG}(S, P, Q)$ has been added relative to a certain subprogram of S and short-circuits some commands in that subprogram. We consider any sequencing of these extra edges; the proof proceeds by induction on the length of this sequence, using Lemma 10 (from right to left) and Proposition 6 (1). \square

The slice graph then represents the entire set of specification-based slices of S , and obtaining the minimal slice is simply a matter of selecting the shortest subsequences using the information in the graph.

Slicing Algorithms. A consequence of the previous result is that the problem of determining the minimal slice of a given program S with respect to the specification (P, Q) can be reduced to determining the minimal control flow graph contained in the slice graph $G = \text{SLCG}(S, P, Q)$.

Consider the case of programs without loops or conditionals, consisting only of atomic commands. Figure 7.3 shows the slice graphs for the two problematic examples presented in Section 7.1. It is clear that for such programs the control flow graph of the minimal slice (i.e. the slice containing the smallest number of atomic commands) can be determined by a standard (unweighted) shortest paths algorithm (basically a breadth-first traversal, executed in linear time on the size of the graph). This CFG contains of course a single path from *START* to *END*.

For programs containing loops, the same algorithm can be used. Following our remarks on slicing subprograms in Section 7.3, determining a minimal slice of a program implies determining the minimal slices of its subprograms, but from the point of view of slice graphs this is irrelevant: when

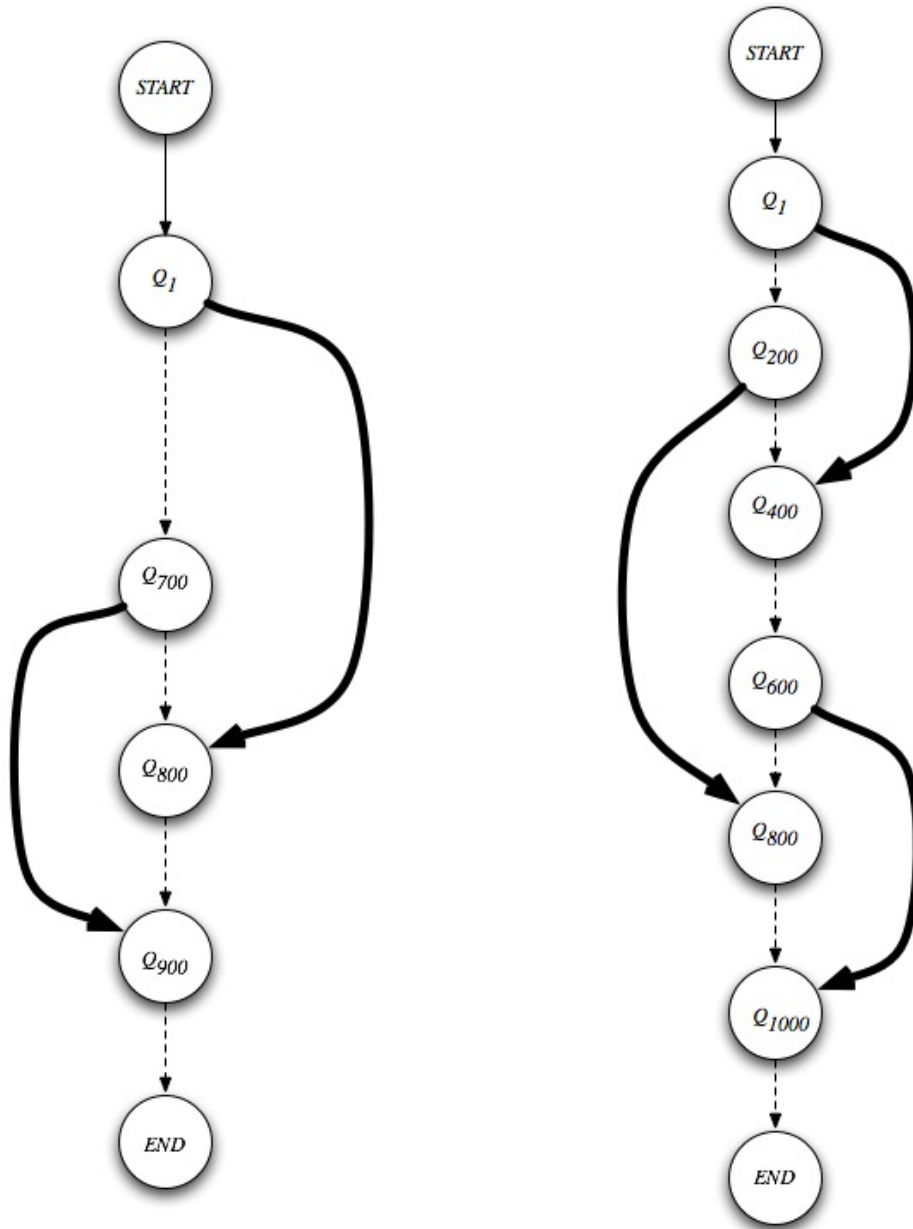


Figure 7.3: Example slice graphs

facing a *while* loop, the shortest path algorithm will have to cross from the *do* node to the *od* node, and will naturally determine the minimal slice of the loop body subprogram.

Conditional commands pose a more substantial problem. Simply applying a shortest paths algorithm would select *one of the branches* of the conditional; what is required is to slice both branches with respect to their local specifications, and then take into account the total number of lines of code of both branches, when slicing the sequence of commands containing this conditional. One way to do this, combining a *weighted* shortest paths algorithm with graph rewriting, is sketched as follows:

1. Assign a weight of 1 to every edge of the slice graph G .
2. For all conditional commands that do not contain any conditional commands as subprograms,
 - (i) run a shortest paths algorithm on the subgraphs of G corresponding to both branches of the conditional, and let $x = 1 + l + r$, where l and r are the sum of the weights of the resulting paths in the *then* and *else* branch respectively;
 - (ii) replace both these subgraphs by a *single edge* with origin *if* and destination *fi*, with weight x ;
3. More conditional commands containing no branching in their subprogram graphs may now have been created; if so, repeat from step 2.

Some mechanism must additionally be used to keep track of the rewritten subgraphs, to allow the slice to be read back from the final graph.

Finally, we stress again that the notion of minimality implicit in this discussion is relative, since it is meant with respect to a slice graph: the proof tool may have failed or timed out in checking some valid condition, and thus an edge that should have been included in the graph is missing; the resulting slice will only be as good as the graph.

Intermediate Assertions. Computing slices in the presence of intermediate assertions as introduced in Section 7.3.3 requires no major modifications in our setting. It suffices to locate in the slice graph the edge (C_k, C_{k+1}) with label (ϕ, ψ) , and replace it by two new edges (C_k, New) and (New, C_{k+1}) with labels (ϕ, R) and (R, ψ) respectively, where *New* is a new node inserted in the graph. The standard algorithm will then compute a slice taking the intermediate condition into consideration.

7.5 Removing Redundant Code

Redundant code is code that does not produce any effect: removing it results in a program that behaves in the same way as the original. Note that it is said “the code does not produce any effect” in the sense of observable effects on the final state. Removing redundant code may of course result in code that is different regarding the execution traces; in particular the resulting code may be faster to execute. A major application of precondition-based slicing is the removal of *conditionally redundant code*, i.e. code that is redundant for executions of the program specified by a given precondition. Naturally, redundant code is a special case of conditionally redundant code (i.e., *true* is considered as precondition).

Examples of redundancies include sequences of instructions like $x := x - 200; x := x + 200$, as in Listing 7.3. Previously in Section 7.1 it was shown that precondition-based slicing with respect to the precondition $x \geq 0$ indeed removed these two instructions. It is however clear that the instructions should be removable also for executions not allowed by this precondition. Let us now consider how this can be done.

A first attempt could be to slice the program with respect to the precondition \top . For Listing 7.3 we would have

$$\begin{aligned}\overline{\text{spost}}_1(S, \top) &= \exists v. x = v + 100 && \equiv \top, \\ \overline{\text{spost}}_2(S, \top) &= \exists v. x = v - 200 && \equiv \top, \\ \overline{\text{spost}}_3(S, \top) &= \exists v. x = v + 200 && \equiv \top\end{aligned}$$

the entire program can now be sliced off, since its calculated postcondition is a valid assertion – not the intended goal. What is missing here is a way to record the initial state, to be able to compare the values of variables in different states using the initial values as a reference. For this purpose one was resort to *auxiliary variables*, that are used in assertions only, not in the code. The use of these variables makes postcondition calculations resemble a *symbolic execution* of the code, in which the values of the variables after the execution of each command are related to the initial values through equality formulas.

Let us slice the same program with respect to the precondition $x = x_0$, where the auxiliary variable x_0 is used to record the initial value of x :

$$\begin{aligned}\overline{\text{spost}}_1(S, x = x_0) &= \exists v. v = x_0 \wedge x = v + 100 && \equiv x = x_0 + 100, \\ \overline{\text{spost}}_2(S, x = x_0) &= \exists v. v = x_0 + 100 \wedge x = v - 200 && \equiv x = x_0 - 100, \\ \overline{\text{spost}}_3(S, x = x_0) &= \exists v. v = x_0 - 100 \wedge x = v + 200 && \equiv x = x_0 + 100\end{aligned}$$

Notice that the precondition does not restrict the set of executions, since x_0 is not a program variable. Since $\models \overline{\text{spost}}_1(S, \top) \rightarrow \overline{\text{spost}}_3(S, \top)$, the statements in lines 2 and 3 of the program can be sliced off, because they are redundant and unnecessary in *any* execution of the program.

```

1 x := y + 2;
  x := y * 2;

```

Listing 7.7: Example for dead code elimination by precondition-based slicing

Two particular forms of redundant code are *unreachable code*, which is not executed, and *dead code*, which is executed but produces no effect, because the final values of variables do not depend on its results. An example of unreachable code is the block S_2 in **if** $10 > 5$ **then** S_1 **else** S_2 ; an example of dead code is the first instruction in Listing 7.7. Unreachable code and dead code elimination are typically part of the optimizations performed by compilers, using control flow and data flow analyses. Specification-based slicing allows for conditional versions of these notions, eliminating code that is unreachable or dead for a given set of executions. Conditional unreachable code elimination was already exemplified with Listing 7.4 in Section 7.1.2 – unreachable code is eliminated because (for the given initial states) its presence does not influence the final state of the program. Precondition-based slicing can thus be used to study the control flow of a program.

Let us now consider an example of (unconditional) dead code elimination. So far slices were identified by checking the validity of implicative formulas involving propagated strongest postconditions. As hinted in the previous section, this technique cannot eliminate all types of redundant code, as this example will also illustrate. Let S be the program in Listing 7.7. Clearly, in every execution of this program, the first statement is dead since its effect is cancelled by the second statement. To slice this program using the precondition $x = x_0 \wedge y = y_0$ we compute the strongest postconditions as follows:

$$\begin{aligned}
\overline{\text{spost}}_0(S, x = x_0 \wedge y = y_0) &= x = x_0 \wedge y = y_0 \\
\overline{\text{spost}}_1(S, x = x_0 \wedge y = y_0) &= \exists v. v = x_0 \wedge y = y_0 \wedge x = y + 2 \\
&\equiv x = y_0 + 2 \wedge y = y_0, \\
\overline{\text{spost}}_2(S, x = x_0 \wedge y = y_0) &= \exists v. v = y_0 + 2 \wedge y = y_0 \wedge x = y * 2 \\
&\equiv x = y_0 * 2 \wedge y = y_0
\end{aligned}$$

Note that, since $\not\models \overline{\text{spost}}_0(S, x = x_0 \wedge y = y_0) \rightarrow \overline{\text{spost}}_1(S, x = x_0 \wedge y = y_0)$, the first statement cannot be sliced off. Clearly, it would be impossible to reach the conclusion that this statement is dead by relating the calculated strongest postconditions only, since $\overline{\text{spost}}_0(S, x = x_0 \wedge y = y_0)$ and $\overline{\text{spost}}_1(S, x = x_0 \wedge y = y_0)$ are calculated without even looking at subsequent commands.

The traditional precondition-based slicing algorithm is then unable to remove all redundant code, specifically when a “look ahead” would be required to reach the conclusion that a given statement can be removed.

```

x := y+2;
2 if (a > 0)
    then x := y*2;
4 else skip;
z := x+1

```

Listing 7.8: Example for redundant code elimination by specification-based slicing

As explained previously, specification-based slicing combines strongest postcondition with weakest precondition computations, and it will now be shown that it can be used to properly eliminate redundant code. Let S be a program with variables x^1, \dots, x^n . In order to remove unnecessary code taking into account every execution of S , it suffices to slice S with respect to the following specification (the x_0^i are auxiliary variables).

$$(x^1 = x_0^1, \dots, x^n = x_0^n, \text{spost}(S, x_1 = x_0^1, \dots, x^n = x_0^n))$$

Following Definition 79 (2), the result will still be a precondition-based slice – the problem of the previous attempt was not in the definition of precondition-based slice, but in the method used to compute these slices.

To illustrate this consider again program in Listing 7.7, and compute a precondition-based slice with respect to $x = x_0 \wedge y = y_0$, followed by a postcondition-based slice with respect to the strongest postcondition:

$$\text{spost}(S, x = x_0 \wedge y = y_0).$$

It was seen before that the first step is unable to remove any statements in this example. The calculated postcondition is

$$Q = \text{spost}(S, x = x_0 \wedge y = y_0) \equiv x = y_0 * 2 \wedge y = y_0$$

Let us now calculate weakest preconditions using the above as postcondition:

$$\begin{aligned} \overline{\text{wprec}}_2(S, Q) &= y * 2 = y_0 * 2 \wedge y = y_0, \\ \overline{\text{wprec}}_1(S, Q) &= y * 2 = y_0 * 2 \wedge y = y_0 \end{aligned}$$

Now since $\models \overline{\text{wprec}}_1(S, Q) \rightarrow \overline{\text{wprec}}_2(S, Q)$, the statement in line 1 can indeed be removed, as would be expected.

Listing 7.8 is a further example of a precondition-based slice that will be calculated as a specification-based slice, following the ideas outlined above (it is taken from [CCL98], see Section 7.6). The idea is to slice this program with respect to the precondition $a > 0$. Clearly the *else* branch is dead, and if it was not already **skip** it would be replaced by **skip** in the computed slice, since it will not be executed with this precondition. The goal here is to illustrate something else: since the *then* branch will be executed, the first

statement in the program will not produce any effect, since the final value of x will be given by the statement in line 3. It is thus a dead statement that should be eliminated.

Let S be the above program and P be $x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0$. In order to eliminate redundant code a slice of this program with respect to the specification $(P, \text{spost}(S, P))$ will be calculated. We start by propagating the precondition forward using strongest postcondition calculations, and then propagate backward the strongest postcondition, using weakest precondition calculations. This is shown in Figure 7.4 (these assertions are also shown in Figure 7.5 in a simplified way). Clearly $\not\models \overline{\text{spost}}_0(S, P) \rightarrow \overline{\text{spost}}_1(S, P)$, but $\models \overline{\text{wprec}}_1(S, P) \rightarrow \overline{\text{wprec}}_2(S, P)$, thus the first command in the program can be eliminated.

These examples further motivate one decision in this chapter: to focus on methods for calculating specification-based slices. Whenever the specification consists of a precondition (resp. postcondition) only, it will be completed by computing the strongest postcondition (resp. weakest precondition) of the program with respect to it. Computing precondition or postcondition-based slices as special cases of specification-based slices allows for a more precise identification of removable statements.

7.6 Related Approaches

In this section other forms of slicing that have some points in common with assertion-based slicing are reviewed.

Semantic Slicing

One of the most successful lines of work in the area of slicing has been conducted by Ward and colleagues. This line has focused on semantic forms of slicing, in the sense that slices are obtained by combining syntactic operations with classic semantics-preserving *program transformations* such as loop unrolling and constant propagation. The results are both practical (a commercially-available workbench has been developed) and theoretical. In particular, the recent paper [War09] provides a clarifying analysis of slicing properties and definitions proposed by different authors (both syntactic and semantic). The PhD work here reported clearly stands on the semantic side, but a fundamental difference with respect to other work on semantic slicing is that it is focused on code annotated with assertions. The slicing criteria are exclusively provided by such assertions.

Conditioned Slicing

Shortly after the definition of postcondition-based slicing by Comuzzi and Hart, Canfora et al [CCL98] introduced the notion of *conditioned slicing*,

Forward propagation of precondition P :

$$\begin{aligned}
\overline{\text{spost}}_0(S, P) &= x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0 \\
\overline{\text{spost}}_1(S, P) &= x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0 \\
\text{spost}(x := y * 2, a > 0 \wedge \overline{\text{spost}}_1(S, P)) &= a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \\
&\quad \wedge a = a_0 \wedge a > 0 \\
\text{spost}(\text{skip}, \neg a > 0 \wedge \overline{\text{spost}}_1(S, P)) &= \neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \\
&\quad \wedge a = a_0 \wedge a > 0 \\
\overline{\text{spost}}_2(S, P) &= (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
\overline{\text{spost}}_3(S, P) &= (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge z = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)
\end{aligned}$$

Backward propagation of postcondition $\overline{\text{spost}}_3(S, P)$:

$$\begin{aligned}
\overline{\text{wprec}}_4(S, P) = \overline{\text{spost}}_3(S, P) &= (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge z = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0) \\
\overline{\text{wprec}}_3(S, P) &= (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
\text{wprec}(x := y * 2, \overline{\text{wprec}}_3(S, P)) &= (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0) \\
\text{wprec}(\text{skip}, \overline{\text{wprec}}_3(S, P)) &= (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge x + 1 = y_0 + 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0) \\
\overline{\text{wprec}}_2(S, P) &= (a > 0 \rightarrow (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)) \\
&\quad \wedge (\neg a > 0 \rightarrow (a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \\
&\quad \wedge x + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge x = y_0 + 2 \wedge y = y_0 \wedge x + 1 = y_0 + 2 + 1 \\
&\quad \wedge a = a_0 \wedge a > 0)) \\
\overline{\text{wprec}}_1(S, P) &= (a > 0 \rightarrow (a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge y * 2 = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge y * 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0)) \\
&\quad \wedge (\neg a > 0 \rightarrow (a > 0 \wedge y + 2 = y_0 * 2 \wedge y = y_0 \wedge \\
&\quad y + 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \wedge a > 0) \\
&\quad \vee (\neg a > 0 \wedge y + 2 = y_0 + 2 \wedge y = y_0 \\
&\quad \wedge y + 2 + 1 = y_0 + 2 + 1 \wedge a = a_0 \wedge a > 0))
\end{aligned}$$

Figure 7.4: Propagated conditions for the program in Listing 7.8

$$\begin{array}{ll}
\overline{\text{spost}}_0(S, P) & \equiv x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0 \\
\overline{\text{spost}}_1(S, P) & \equiv x = y_0 + 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \wedge a > 0 \\
\text{spost}(x := y * 2, a > 0 \wedge \overline{\text{spost}}_1(S, P)) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \\
\text{spost}(\text{skip}, \neg a > 0 \wedge \overline{\text{spost}}_1(S, P)) & \equiv \perp \\
\overline{\text{spost}}_2(S, P) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = z_0 \wedge a = a_0 \\
\overline{\text{spost}}_3(S, P) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \\
& \quad \wedge a = a_0 \\
\\
\overline{\text{wprec}}_4(S, P) = \overline{\text{spost}}_3(S, P) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge z = y_0 * 2 + 1 \\
& \quad \wedge a = a_0 \\
\overline{\text{wprec}}_3(S, P) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \\
& \quad \wedge a = a_0 \\
\text{wprec}(x := y * 2, \overline{\text{wprec}}_3(S, P)) & \equiv a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
& \quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \\
\text{wprec}(\text{skip}, \overline{\text{wprec}}_3(S, P)) & \equiv a > 0 \wedge x = y_0 * 2 \wedge y = y_0 \wedge x + 1 = y_0 * 2 + 1 \\
& \quad \wedge a = a_0 \\
\overline{\text{wprec}}_2(S, P) & = a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
& \quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0 \\
\overline{\text{wprec}}_1(S, P) & = a > 0 \wedge y * 2 = y_0 * 2 \wedge y = y_0 \\
& \quad \wedge y * 2 + 1 = y_0 * 2 + 1 \wedge a = a_0
\end{array}$$

Figure 7.5: Simplified conditions for the program in Listing 7.8

together with a tool to calculate such slices. Similarly to precondition-based slicing, conditioned slicing uses preconditions as a means to specify a set of initial states for computing a forward slice. The main points to understand about conditioned slicing are

1. The precondition is used in combination with traditional slicing techniques based on dependency analysis. Code will be removed either because it is unreachable (not executed when the program is started in a state in which the precondition holds), or because it is dead (the precondition eliminates dependencies involving it). Consider the following example from Canfora's paper:

```

1 x := y+2;
  if (a > 0)
3   x := y*2;
  z := x+1

```

A conditioned slice of this program based on any precondition P such that $\models P \rightarrow a > 0$ results in line 1 being eliminated, since line 3 will certainly be executed and cancel the effect of line 1. This example shows a fundamental difference between conditioned slicing and earlier notions of slicing exclusively based on control and data dependencies. Clearly static dependencies alone cannot be used to implement conditioned slicing, since the instruction in line 4 depends on all previous instructions. The algorithm proposed by the authors is based on *symbolic execution*, which allows for the relevant dependency paths to be identified. A theorem prover is called externally to guide the symbolic execution.

2. In the context of traditional, dependency-based slicing, there are two

standard types of forward slicing: *static*, which considers every possible execution (i.e. all initial states), and *dynamic*, which is concerned with a single execution (a concrete initial state) of the program. The latter can be generalized to cope with a set of concrete executions, but an interesting aspect of conditioned slicing is that it subsumes all these notions, since a characterization of the set of initial states by a first-order condition can be used to admit any initial state (if the condition is \top), or just a concrete initial state (if the condition is a conjunction of equality formulas, each equating a program variable to a constant), or any other intermediate set of initial states.

3. The similarities between precondition-based slicing and conditioned slicing should be clear: even though the latter is based on dependencies and the former on weakest preconditions and strongest postconditions, both are capable of eliminating conditionally unreachable and conditionally dead code. These are examples of code that is redundant with respect to a given precondition, but note that the notion of redundancy is different in both cases: whereas in precondition-based slicing this is code that, if removed, results in a program whose strongest postcondition will not be weakened with respect to the initial program, in conditioned slicing this is code that does not contribute to the values of a given set of variables. Precondition-based slicing removes other forms of redundancy that conditioned slicing cannot remove, since they can only be detected at a semantic level. Listing 7.4 shows a good example to illustrate this point: while conditioned slicing with $x \geq 0$ would eliminate the *else* branch, it would not remove the two assignment commands inside the *then* branch, which are removed by precondition-based slicing.
4. Conditioned slicing criteria are not however limited to a precondition P : a slicing criterion consists additionally of a subset X of the program variables, as well as a specific program line k . The program statements eliminated are those that do not affect the value of any variable in X at line k , for executions starting in states satisfying P . Precondition-based slicing does not subsume conditioned slicing, since it does not take into account these criteria, inherited from standard dependency-based forms of slicing.
5. For conditioned slicing criteria that focus on the final state of the program (i.e. k is the last line), precondition-based slicing can be said to be a stronger form of slicing than conditioned slicing, since it eliminates code using semantic criteria that cannot be expressed in terms of dependencies.

Backward Conditioned Slicing

Backward conditioning was introduced by Fox and colleagues [FDHH01] as the symmetric notion of conditioned slicing. A slicing criterion includes a postcond Q that is used in the following way: statements whose presence *forces* $\neg Q$ to hold in the final state (i.e. if they are present $\neg Q$ will hold after every execution) are removed.

The technique is intended as the dual of conditioned slicing: whereas (forward) conditioned slicing eliminates the code that will surely not be executed when the given precondition holds, backward conditioned slicing eliminates the code that cannot be executed if the given postcondition is to be satisfied in the final state, i.e. it eliminates statements that prevent the given postcondition from being true. The technique is introduced with program comprehension as main application. The authors also propose an algorithm for implementing backward conditioned slicing, based on symbolic execution and an external theorem prover.

A second paper [HHF⁺01] combines forward and backward conditioned slicing, based on a precondition P and a postcondition Q : it eliminates code that leads to Q being false when the program is executed in states satisfying P . The motivation of the latter work is the application to program verification. The idea here is that in order to check if a program is correct w.r.t. a specification (P, Q) , one may compute its conditioned slice w.r.t. $(P, \neg Q)$. If the program is correct this slice will be *empty*, since all execution paths lead to Q being true, and all instructions will thus be removed. If the program is not correct, the instructions that remain in the slice are those that *may for some initial states* lead to $\neg Q$ being true. Such instructions should carefully be considered since they are directly contributing to the program being incorrect.

This forward/backward form of conditioned slicing cannot be formulated as specification-based slicing with respect to a specification. While a specification-based slice S' of program S with respect to (P, Q) is correct with respect to (P, Q) , a conditioned slice S'' with respect to (P, Q) is characterized by *not being correct* with respect to $(P, \neg Q)$. Another way to put this is that while we require $\text{VCG}^w(P, S', Q)$ to be *valid*, $\text{VCG}^w(P, S'', Q)$ must instead be *satisfiable*. For instance the command $x := x + 10$ with precondition $x > 10$ and postcondition $x \leq 20$ should be sliced to **skip**, since $\models \text{VCG}^w(x > 10, x := x + 10, \neg(x \leq 20))$ and $\not\models \text{VCG}^w(x > 10, \text{skip}, \neg(x \leq 20))$, i.e. the formulas $\text{VCG}^w(x > 10, \text{skip}, x \leq 20)$ are satisfiable.

Chapter 8

Contract-based Slicing

*Every law is a contract between
the king and the people and
therefore to be kept.*

John Selden, 1584-1654

The main goal of this chapter is to attempt to answer the following question:

How can assertion-based slicing, formally introduced in the previous chapter, be applied in the context of a multi-procedure program?

Since a program is a set of procedures carrying their own contract specifications, it makes sense to investigate how the contract information can be used to produce useful slices at the level of individual procedures, and globally at the level of programs.

It will be shown that given any assertion-based slicing algorithm, a *contract-based slice* can be calculated by slicing the code of each individual procedure independently with respect to its contract (this will be called an *open slice*), or taking into consideration the calling contexts of each procedure inside a program (which will be called a *closed slice*). Both notions are studied, and then a more general notion of contract-based slice, which encompasses both open and closed slices as extreme cases, is introduced.

Throughout the chapter, it will be considered that a program is a non-empty set of mutually recursive procedure definitions that share a set of global variables, following the principles introduced in Chapter 3. Operationally, an entry point would have to be defined for each such program, but that is not important for the current purpose. For the sake of simplicity only *parameterless procedures* will be considered. A shared set of global variables are used for input and output, but the ideas presented here could

be adapted to cope with parameters passed by value or reference, as well as return values.

Structure of the Chapter. Sections 8.1 and 8.2 introduce contract-based slicing in their more specific (open and closed) and general forms respectively; Section 8.3 then shows how a contract-based slicing algorithm (working at the inter-procedural level) can be synthesized from any given assertion-based slicing algorithm (working at the intra-procedural level).

8.1 Open / Closed Contract-based Slicing

In this section, both notions of *open* and *closed* contract-based slicing are introduced.

The first approach consists in simply slicing each procedure based on its own contract information. The idea is to eliminate all extraneous code that may be present and does not contribute to making the procedure fulfill its contract.

Definition 83 (Open Contract-based Slice). *Given programs Π , Π' such that $\models \text{Verif}(\Pi)$ and $\mathbf{PN}(\Pi) = \mathbf{PN}(\Pi')$, we say that Π' is an open contract-based slice of Π , written $\Pi' \triangleleft_o \Pi$, if for every procedure $\mathbf{p} \in \mathbf{PN}(\Pi)$ the following holds: $\mathbf{pre}_{\Pi'}(\mathbf{p}) = \mathbf{pre}_{\Pi}(\mathbf{p})$; $\mathbf{post}_{\Pi'}(\mathbf{p}) = \mathbf{post}_{\Pi}(\mathbf{p})$; and*

$$\mathbf{body}_{\Pi'}(\mathbf{p}) \triangleleft_{(\mathbf{pre}(\mathbf{p}), \mathbf{post}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$$

i.e. the body of each routine in Π' is a specification-based slice (with respect to its own annotated contract) of that routine in Π .

As expected, open contract-based slicing produces correct programs:

Proposition 8. *If $\models \text{Verif}(\Pi)$ and $\Pi' \triangleleft_o \Pi$ then $\models \text{Verif}(\Pi')$.*

Proof. Straightforward from Definitions 83 and 51. □

Let `slice` be a specification-based slicing algorithm that given a block S and a specification (P, Q) produces a slice S' of S with respect to (P, Q) , i.e., $\text{slice}(S, P, Q) \triangleleft_{(P, Q)} S$. It is straightforward to lift it to an algorithm that calculates contract-based slices.

Definition 84. *Let Π be a program, $\mathbf{p} \in \mathbf{PN}(\Pi)$ and*

$$\begin{aligned} \text{procslice}_o(\mathbf{p}) &\doteq \mathbf{pre}(\mathbf{pre}_{\Pi}(\mathbf{p})) \\ &\quad \mathbf{post}(\mathbf{post}_{\Pi}(\mathbf{p})) \\ &\quad \mathbf{proc} \mathbf{p} = \text{slice}(\mathbf{body}_{\Pi}(\mathbf{p}), \mathbf{pre}_{\Pi}(\mathbf{p}), \mathbf{post}_{\Pi}(\mathbf{p})) \end{aligned}$$

Then $\text{progslice}_o(\Pi) \doteq \{ \text{procslice}_o(\mathbf{p}) \mid \mathbf{p} \in \mathbf{PN}(\Pi) \}$.

Proposition 9. *For any program Π such that $\models \text{Verif}(\Pi)$, $\text{progslice}_o(\Pi) \triangleleft_o \Pi$.*

Proof. Straightforward from Definitions 83 and 84. \square

As was the case with assertion-based slicing, one may want to calculate open contract-based slices just to make sure that a program (already proved correct) does not contain irrelevant code. This notion of slice of a program assumes that all the procedures are public and may be externally invoked – the program is *open*. But consider now the opposite case of a program whose procedures are only invoked by other procedures in the same program, which we thus call a *closed* program (this makes even more sense if one substitutes *class* for *program* and *method* for *procedure* in this reasoning). In this situation, since the set of callers of each procedure is known in advance (it is a subset of the procedures in the program), it is possible that *weakening* the procedures' contracts may still result in a correct program, as long as the assumptions required by each procedure call are all still respected. In other words the procedures may be doing more work than is actually required. Such a program may then be sliced in a more aggressive way, defined as follows.

Definition 85 (Closed Contract-based Slice). *Let Π, Π' be programs such that $\models \text{Verif}(\Pi)$ and $\mathbf{PN}(\Pi) = \mathbf{PN}(\Pi')$. Π' is a closed contract-based slice of Π , written $\Pi' \triangleleft_c \Pi$, if $\models \text{Verif}(\Pi')$ and additionally for every procedure $p \in \mathbf{PN}(\Pi)$*

1. $\models \mathbf{pre}_{\Pi'}(p) \rightarrow \mathbf{pre}_{\Pi}(p)$ (strengthening the precondition);
2. $\models \mathbf{post}_{\Pi}(p) \rightarrow \mathbf{post}_{\Pi'}(p)$ (weakening the postcondition); and
3. $\mathbf{body}_{\Pi'}(p) \triangleleft_{(\mathbf{pre}_{\Pi'}(p), \mathbf{post}_{\Pi'}(p))} \mathbf{body}_{\Pi}(p)$ (the body of the procedure is an assertion-based slice of the original one with respect to the new specification).

In general weakening the contracts of some procedures in a correct program may result in an incorrect program, since the correctness of each procedure may depend on the assumed correctness of other procedures; thus the required condition $\models \text{Verif}(\Pi')$ in the definition of closed contract-based slice.

As a very simple example, consider a program Π containing the procedure **FactFib** where $\mathbf{pre}_{\Pi}(\text{FactFib})$ is $x \geq 0$ and $\mathbf{post}_{\Pi}(\text{FactFib})$ is $f_a = \text{fact}(x) \wedge f_i = \text{fib}(x)$, where the logical functions **fact** and **fib** capture axiomatically the Factorial and Fibonacci functions. If every call made in Π to this procedure only makes use of the calculated value f_a of factorial, then the program Π' can contain a weaker contract for **FactFib**, with $\mathbf{post}_{\Pi'}(\text{FactFib})$ being $f_a = \text{fact}(x)$. Since the resulting program would still be correct and $\models \mathbf{post}_{\Pi}(\text{FactFib}) \rightarrow \mathbf{post}_{\Pi'}(\text{FactFib})$ (and the preconditions are the same), Π' would be a closed contract-based slice of Π .

8.2 Contract-based Slicing: General Case

Clearly the notion of closed contract-based slice admits trivial solutions: since all contracts can be weakened, any precondition (resp. postcondition) can be set to *false* (resp. *true*), and thus any procedure body can be sliced to **skip**. A more interesting and realistic notion is obtained by fixing a subset of procedures of the program, whose contracts must be preserved. All other contracts may be weakened as long as the resulting program is still correct.

Definition 86 (Contract-based Slice). *Let Π, Π' be programs such that $\models \text{Verif}(\Pi)$, $\mathbf{PN}(\Pi) = \mathbf{PN}(\Pi')$ and $\mathcal{S} \subseteq \mathbf{PN}(\Pi)$; Π' is a contract-based slice of Π , written $\Pi' \triangleleft_{\mathcal{S}} \Pi$, if the following all hold:*

1. $\models \text{Verif}(\Pi')$.
2. for every procedure $\mathbf{p} \in \mathcal{S}$,
 - $\mathbf{pre}_{\Pi'}(\mathbf{p}) = \mathbf{pre}_{\Pi}(\mathbf{p})$
 - $\mathbf{post}_{\Pi'}(\mathbf{p}) = \mathbf{post}_{\Pi}(\mathbf{p})$
 - $\mathbf{body}_{\Pi'}(\mathbf{p}) \triangleleft_{(\mathbf{pre}_{\Pi}(\mathbf{p}), \mathbf{post}_{\Pi}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$
3. for every procedure $\mathbf{p} \in \mathbf{PN}(\Pi) \setminus \mathcal{S}$,
 - $\models \mathbf{pre}_{\Pi'}(\mathbf{p}) \rightarrow \mathbf{pre}_{\Pi}(\mathbf{p})$;
 - $\models \mathbf{post}_{\Pi}(\mathbf{p}) \rightarrow \mathbf{post}_{\Pi'}(\mathbf{p})$; and
 - $\mathbf{body}_{\Pi'}(\mathbf{p}) \triangleleft_{(\mathbf{pre}_{\Pi'}(\mathbf{p}), \mathbf{post}_{\Pi'}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$

This notion is very adequate to model the process of slicing when applied to code reuse. When the program (or class) Π is reused, some of its procedures may not need to be public, since they will not be externally invoked (but they may be invoked by other, public procedures in the program). In this case the contracts of the private procedures may be weakened according to their usage inside the program, i.e. the actual required specification for each private procedure may be calculated from the set of internal calls, since it is guaranteed that no external calls will be made to private procedures. The user may then want to reflect this in the annotated contracts, in order to produce a contract-based slice stripped of the redundant code. Private procedures whose contracts are not required internally may indeed see their bodies sliced to **skip**.

Even for closed programs this notion makes more sense than the idealized Definition 85. Since its procedures are not invoked externally from other programs' procedures, every closed program will naturally have a main procedure to be executed as an entry point, whose contract is fixed (cannot be weakened).

8.3 A Contract-based Slicing Algorithm

Again any specification-based slicing algorithm (at the command block level) can be used for calculating contract-based slices according to Definition 86. A contract-based slice of program Π can be calculated by analyzing Π in order to obtain information about the actual preconditions and postconditions that are required of each procedure call, and merging this information together. Specifically, it is possible to calculate for each procedure the disjunction of all required preconditions and the conjunction of all required postconditions; this potentially results in a weaker contract with respect to the annotated contract of the procedure, which can thus be used to slice that procedure.

To implement this idea for a given program Π we consider a preconditions table T_{pre} that associates to each procedure $\mathbf{p} \in \mathbf{PN}(\Pi)$ a precondition, initialized with $T_{pre}[\mathbf{p}] = \perp$, and a postconditions table T_{post} that associates to each procedure $\mathbf{p} \in \mathbf{PN}(\Pi)$ a postcondition, initialized with $T_{post}[\mathbf{p}] = \top$. The algorithm executes the following steps to produce $\text{progslice}_S(\Pi)$ for a given set of procedures $S \subseteq \mathbf{PN}(\Pi)$.

1. Calculate $\text{Verif}(\Pi)$ based on the weakest preconditions and while doing so, for every invocation of the form $\text{wprec}(\text{call } \mathbf{p}, Q)$ set $T_{post}[\mathbf{p}] := T_{post}[\mathbf{p}] \wedge Q$.
2. Calculate $\text{Verif}(\Pi)$ based on the strongest postconditions and while doing so, for every invocation of the form $\text{spost}(\text{call } \mathbf{p}, P)$ set $T_{pre}[\mathbf{p}] := T_{pre}[\mathbf{p}] \vee P$.
3. For $\mathbf{p} \in \mathbf{PN}(\Pi) \setminus S$ let

$$\begin{aligned} \text{procslice}_S(\mathbf{p}) \quad \doteq \quad & \text{pre } T_{pre}[\mathbf{p}] \\ & \text{post } T_{post}[\mathbf{p}] \\ & \text{proc } \mathbf{p} = \text{slice}(\text{body}(\mathbf{p}), T_{pre}[\mathbf{p}], T_{post}[\mathbf{p}]) \end{aligned}$$

4. Then

$$\begin{aligned} \text{progslice}_S(\Pi) \quad = \quad & \{ \text{procslice}_o(\mathbf{p}) \mid \mathbf{p} \in S \} \cup \\ & \{ \text{procslice}_S(\mathbf{p}) \mid \mathbf{p} \in \mathbf{PN}(\Pi) \setminus S \} \end{aligned}$$

To see that for any program Π such that $\models \text{Verif}(\Pi)$, $\text{progslice}_S(\Pi) \triangleleft_S \Pi$, it suffices to show that the postconditions calculated for the procedures in the slice are weaker than the initial postconditions, and that the program produced has valid verification conditions. The first part is a consequence of the contents of table T_{post} after step 1 one of the algorithm and the fact that the verification conditions $\text{Verif}(\Pi)$ are all valid.

The second part is relatively easy for this particular algorithm because the preconditions in the contracts are preserved. Since the verification conditions of the initial program are all valid, it suffices to prove that the verification conditions of the slice Π' are entailed by those of the initial program Π . This is not an immediate consequence for an arbitrary weakening of the postconditions in the contracts, but it is true for the particular postconditions that are present in the sliced program's contracts, since for every calculation $\text{wprec}(\text{call } \mathbf{p}, Q)$ or $\text{VC}^w(y := \text{fcall } \mathbf{f}(\bar{x}), Q)$, the postconditions $\text{post}_{\Pi'}(\mathbf{p})$ and $\text{post}_{\Pi'}(\mathbf{f})$ are of the form $Q_1 \wedge \dots \wedge Q_k$, where $Q_j = Q$ for some $j \in \{1, \dots, k\}$.

Note that step 1 (or 2) can be skipped, in which case $T_{\text{post}}[\mathbf{p}]$ (resp. $T_{\text{pre}}[\mathbf{p}]$) should be set to $\text{post}(\mathbf{p})$ (resp. $\text{pre}(\mathbf{p})$), and slicing will be less aggressive, based on preconditions or postconditions only.

In Chapter 9, page 269, the application of this contract-based slicing algorithm will be illustrated with a program containing a procedure \mathbf{p} that calculates a number of different outputs given an input array; this program is reused in a context in which no external calls are made to \mathbf{p} , and two internal procedures do call \mathbf{p} , but some of the information computed by \mathbf{p} is not required by either of the calls. Then some statements of \mathbf{p} can possibly be sliced off.

Chapter 9

GamaSlicer tool

*It doesn't matter how beautiful
your theory is, it doesn't matter
how smart you are. If it doesn't
agree with experiment, it's
wrong.*

Attributed to Richard Phillips
Feynman, 1918-1988

This chapter introduces **GamaSlicer**, a tool whose main goal is to illustrate that all the concepts introduced in previous chapters work in practice. **GamaSlicer** includes both traditional and interactive verification functionality (it is a Verification Conditions Generator) and also a highly parameterizable semantic slicer for Java programs annotated in JML. It includes all the published algorithms (precondition-based slicing, postcondition-based slicing, specification-based slicing and their variants) as well as the new ones introduced in Chapters 7 and 8. On one hand, the tool can be useful to detect possible errors in the program through the use of the interactive verification of the program. On the other hand, the tool can be useful for comparing the effectiveness of slicing algorithms; it allows users to perform different workflows, such as:

- First verifying a program with respect to a given specification; then slicing it with respect to that specification to check if there are irrelevant commands (with respect to that specification).
- From a verified program, producing a specialization by weakening the specification and slicing the program with respect to the weaker specification. This may be useful in a program reuse context, in which a weaker contract is required for a component than the actually implemented contract.

In this chapter, examples of the different features of GamaSlicer will be shown as well as the results produced. Concerning the novelties of this tool, at least the following can be listed:

- A VCGen, where the user can control in a step-by-step fashion the generation of the proof obligations.
- An interactive VCGen, where the user can control which parts of the program he wants to verify.
- A slicer that implements the previous published algorithms using specifications to slice a program, and the new algorithms and slicing definitions proposed along this document.
- A slicing algorithm animator, where the user can have also control over the animation and see the effects of each step.

Structure of the chapter. Section 9.1 introduces the architecture of GamaSlicer and discussed some concerning its design. In Section 9.2, through an example, it is explained how works the process of verifying a program in a interactive way. In Section 9.3, through several examples, the different features offered by GamaSlicer with respect to slicing are illustrated.

9.1 GamaSlicer Architecture

The architecture of GamaSlicer¹, inspired by that of a compiler (or generally speaking a language processor), is depicted in Figure 9.1. It is composed of the following blocks: a Java/JML front-end (a parser and an attribute evaluator) — see Appendix A for more details about JML; a verification conditions generator; a slicer; and a labeled control flow graph (LCFG) visualizer/animator for both the Verification Graphs and Slice Graphs.

Since the underlying logic of slicing algorithms as well as the VCGen is first-order logic, the tool outputs proof obligations written in the SMT-Lib (Satisfiability Modulo Theories library) language. It was chosen SMT-Lib since it is nowadays the language employed by most provers used in program verification, including, among many others, Z3 [dMB08a], Alt-Ergo [CCK06], and Yices [DdM06] — see Appendix B for more details about SMT.

After uploading a file containing a piece of Java code together with a JML specification, the code is recognized by the front-end (an analyzer produced automatically from an attribute grammar with the help of ANTLR parser generator [Par07]), and it is transformed into an Abstract Syntax Tree (AST — Definition 1). The choice of an Intermediate Representation

¹Version 1.0 is available online at <http://gamaepl.di.uminho.pt/gamaslicer>. Version 2.0 was released as a desktop version available for download at the same URL.

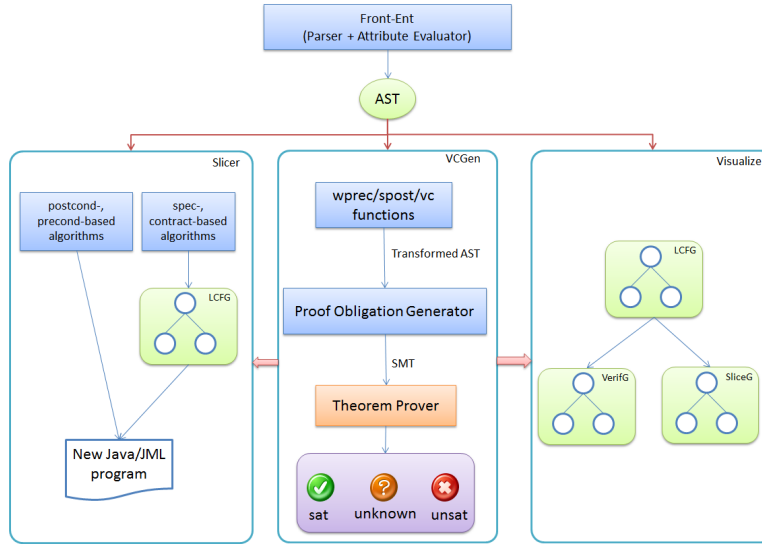


Figure 9.1: GamaSlicerarchitecture

(an AST) as the core makes the tool language independent, that is, all the features can be applied to another source language if a front-end for it is built. This way, all the modules (VCGen, slicer and visualizer/animater) work over this representation traversing it and transforming it through pattern matching. Some of the patterns currently supported are displayed in Table 9.1 — please notice the similarity between these patterns and the abstract syntax of commands defined in Figure 3.3 (Chapter 3).

During this first step, the analysis and intermediate representation of the source code, an Identifier Table is also built.

The intermediate information that becomes available at each step is displayed in a window distributed by nine main tabs (as can be seen in Fig-

SKIP ASSIGN <i>left right</i> STATEMENTS $S_1 \dots S_n$ IF (CONDITION <i>b</i>) (THEN S_t) (ELSE S_f) WHILE (LOOP_INVARIANT <i>i</i>) (LOOP_TEST <i>b</i>) (STATEMENTS <i>S</i>) CALL (NAME <i>f</i>) (ARGUMENTS $arg_1 \dots arg_n$)	the skip command assignment command sequence of commands conditional statement loop command annotated with an invariant call of a procedure
--	---

Table 9.1: Some of the patterns recognized by GamaSlicer

ure 9.2):

- *Tab 1*: contains the Java/JML source program to be analyzed.
- *Tab 2*: contains the syntax tree (AST) generated by the front-end.
- *Tab 3*: contains the Identifier Table built during the analysis phase.
- *Tab 4, Tab 5 and Tab 6*: these three tabs are used to verify the correctness of code with respect to contracts. The first two are used during the standard verification process through a VCGen.

In tab 4, the rules applied along the generation of the verification conditions are displayed in tree format.

In tab 5, the generated SMTcode is shown; also a table with the verification status of each formula (**sat**, **unsat**, **unknown**) after calling a theorem-prover will be displayed.

Tab 6 is used to perform an interactive verification of a program.

- *Tab 7*: in this tab, the user can select which slicing algorithm wants to apply. After applying contract-based slicing algorithms to the original program, it will contain the new program produced by the slicer; notice that useless statements identified by the slicer are not actually removed, but shown in red and strike-out style.
- *Tab 8*: displays the labeled control flow graph (LCFG) as the visual representation of the program, giving an immediate and clear perception of the program complexity, with its procedures and the relationships among them.
- *Tab 9*: in this tab, the user can select which algorithm to animate. The algorithm selected will be animated through the use of the LCFG, allowing the user to control the animation process.

The first version of GamaSlicer was built as a web-based tool. The main idea was to make it as an online laboratory available for everyone and freeing the user of any kind of installation. This online laboratory is still available at <http://gamaepl.di.uminho.pt/gamaslicer/> but does not include all the features of the last version.

The main challenge found was when we needed to include the visualizer/animator. There are some graph libraries available web-based, but they either do not allow any kind of customization (addition of edges on a graph, shape changing, etc) or they were not compatible with the framework so far developed.

As a first approach, a graph library was developed based on Dot language, from Graphviz², for Silverlight [Mac10]. A first version of this library is available at <http://dot2silverlight.codeplex.com/>. However,

²<http://graphviz.org/>

this approach proved to be not so good, since Silverlight is not fast in rendering the graph and it does not work well in all browsers and operation systems (OS).

This way, it was decided to upgrade GamaSlicer to a desktop version, since there were interesting graph libraries to use (different libraries were tried, and the one that fitted better for our purposes was **Graph#**³ — the source code is available and thus allows to do any kind of customization). With this upgraded version, the user has of course to install the tool but it is faster than the first version (currently, only Windows OS is available).

Next sections detail the three main features of the tool: VCGen, Slicer and Animator/Visualizer.

9.2 Verifying

As mentioned previously, GamaSlicer provides two modes for verifying the correction of a given program according to its JML contract. First, the traditional VCGen approach is described (based on two steps, the generation of a first-order logic formula and its verification by a prover); then, the interactive version, an innovative result of this PhD work based on verification graphs, is presented.

Traditional approach As discussed in Chapter 3, the verification infrastructure consists of a program logic used to assert the correctness of individual procedures, from which a VCGen algorithm is derived.

With respect to the traditional approach, GamaSlicer allows the verification of a program using the standard method that relies on the algorithm that uses the weakest precondition strategy (Definition 51 on page 74) or using the one that uses the strongest postcondition strategy (Definition 52 on page 75).

To verify a program, after uploading a Java program properly annotated in JML, it is possible to invoke the GamaSlicer VCGen module.

As explained above, this VCGen is implemented as a tree-walker evaluator that traverses the AST to generate the verification conditions, using a tree-pattern matching strategy. Before applying any tree pattern matching rules, a tree traversal is done to look for methods with contracts; this search returns a set of subtrees, and the VCGen algorithm is applied to each one. Through the tree pattern matching process, each time a tree matches a pattern, it is transformed and marked as *visited*.

At the end, the transformed AST (now with verification conditions attached to the nodes) is traversed by the SMT code generator (another tree-walker evaluator) to produce SMT proof obligations.

³<http://graphsharp.codeplex.com/>

```

2  /*@
   @ predicate isfact(integer n, integer r);
   @*/
4
6  /*@ axiom isfact0:
   @    isfact(0,1);
   @*/
8
10 /*@ axiom isfactn:
   @    (\forall int n, f;
   @    isfact (n-1,f) ==> isfact(n,f*n));
   @*/
12
14 /*@ axiom factfunctional:
   @    (\forall int n, f1, f2;
   @    isfact(n,f1) ==> isfact(n,f2) ==> f1==f2);
   @*/
16
18
20 public class Factorial1 {
21     /*@ requires n >= 0;
22     @ ensures isfact(n, \result);
23     @*/
24     private int factf (int n)
25     {
26         int f = 1, i = 1;
27
28         //@ loop_invariant i <= n+1 && isfact(i-1,f);
29         while (i <= n) {
30             f = f * i;
31             i = i + 1;
32         }
33         return f;
34     }
35 }

```

Listing 9.1: Factorial program annotated in JML

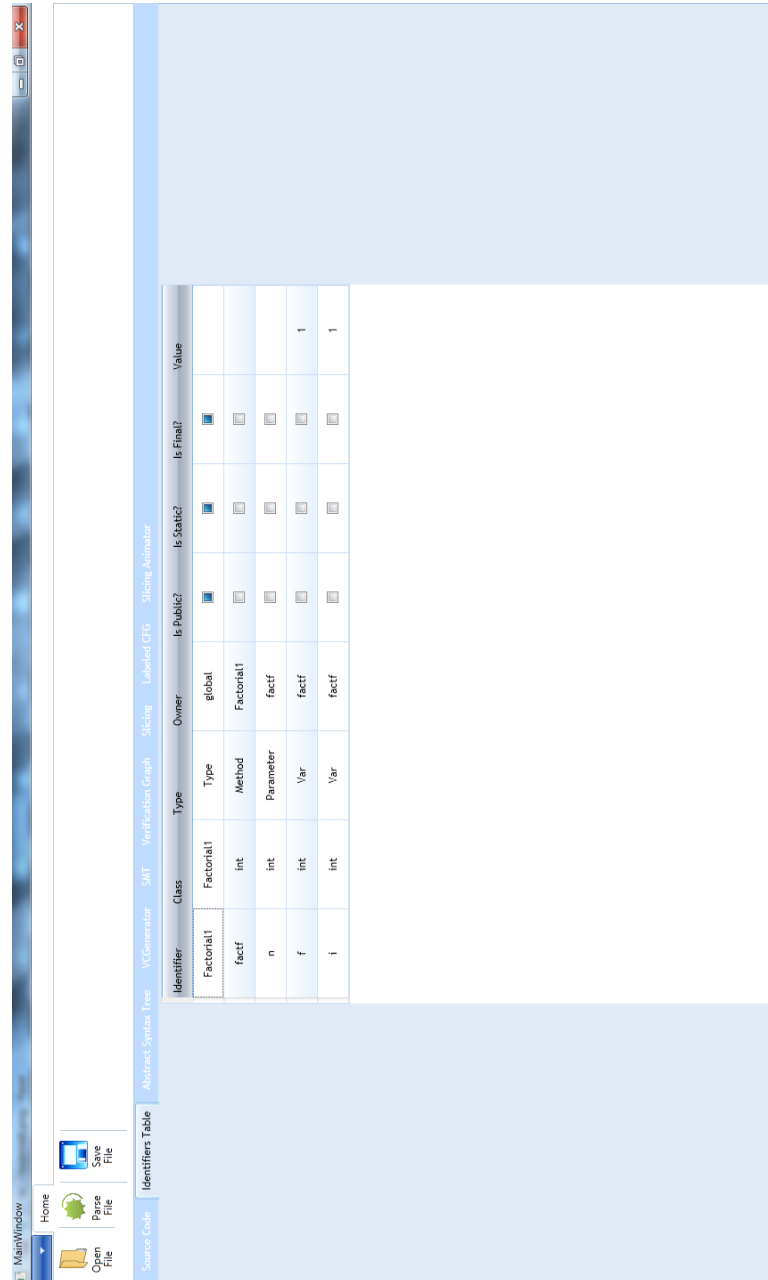
GamaSlicer allows the user to follow the generation of this set of verification conditions in a step-by-step fashion.

As an example, please consider that we have the `Factorial` program properly annotated — Listing 9.2 and Figure 9.2.

Once this program is successfully parsed, it becomes available, as previously explained, the Identifier Table (Figure 9.3) and the AST (Figure 9.4). At this point we can move to the VCGen tab and go step-by-step through the verification process. The first step is to choose the strategy to use on the generation of the verification conditions (using a weakest precondition or a strongest postcondition strategy). Once this generation starts, the sequence of rules applied to the program are presented in a tree form in order to make easier the comprehension of the process (Figure 9.5) — the arguments used in the weakest precondition function as well as in the auxiliary function are shown too. When there are no more rules to apply, it is possible to generate the set of verification conditions in SMT-LIB language, which will be displayed in the SMT tab. At this moment, we can check the validity of these VCs calling the automatic theorem prover (currently Z3 theorem prover is being used). At the end, the results outputted by the prover are displayed in a tabular form (see bottom of Figure 9.6). Analyzing these results it is possible to conclude if the program is correct or not (in this case the `Factorial` program is correct).



Figure 9.2: Factorial program annotated in JML



Identifier	Class	Type	Owner	Is Public?	Is Static?	Is Final?	Value
Factorial()	Factorial()	Type	global	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
factf	int	Method	Factorial()	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
n	int	Parameter	factf	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
f	int	Var	factf	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1
i	int	Var	factf	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1

Figure 9.3: Identifier Table of the Factorial program in Listing 9.1

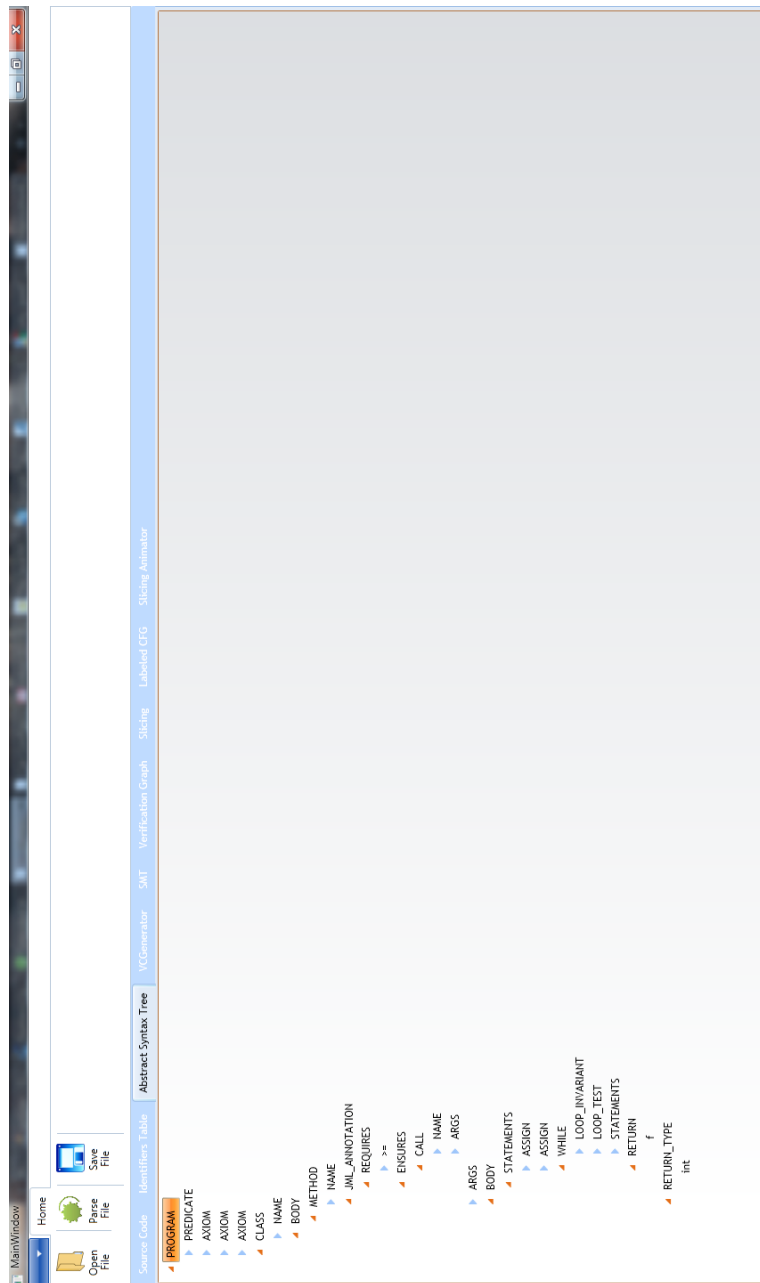
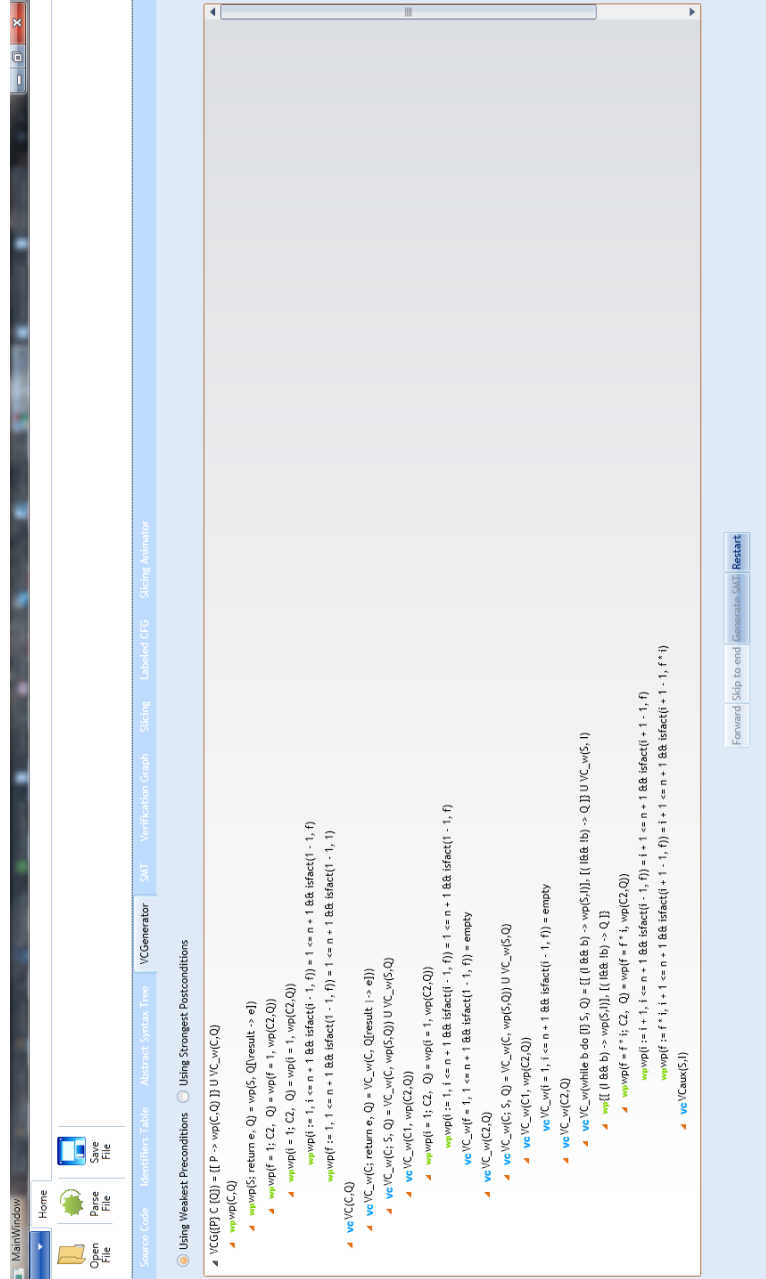


Figure 9.4: AST of the Factorial program in Listing 9.1



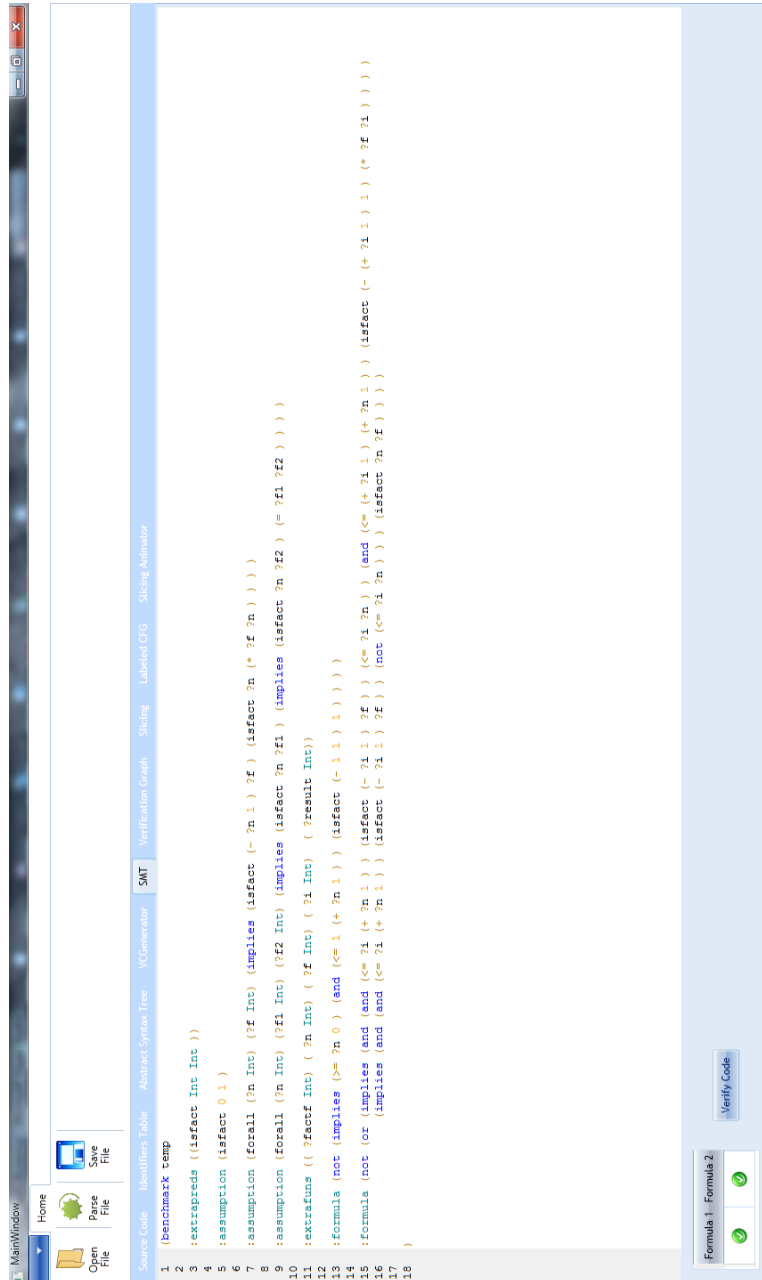


Figure 9.6: Verification Conditions for the Factorial program

```

1 public class UKTaxesCalculation
2 {
3     public int age, income;
4     public int personal, t;
5
6     /*@ requires (age >= 18);
7        ensures (personal > 5750);
8     @*/
9     public void TaxesCalculation()
10    {
11        if (age >= 75) { personal = 5980; }
12        else if (age >= 65) { personal = 5720; }
13        else { personal = 4335; }
14
15        if ((age >= 65) && (income > 16800))
16        {
17            t = personal - ((income - 16800) / 2);
18            if (t > 4335) { personal = t + 2000; }
19            else { personal = 4335; }
20        }
21    }
22 }

```

Listing 9.2: Class TaxesCalculation

Interactive approach Chapter 6 introduced the notion of control flow graph annotated with pairs of assertions (LCFG) and how these graphs could be used to conduct the verification process in a more interactive way. These graphs were called *verification graphs*.

A LCFG is nothing more than a control flow graph annotated with assertions (the *edge conditions*) from which we can pick different sets whose validity is sufficient to guarantee the correctness of the procedure. Each particular set corresponds to one particular verification strategy, mixing the use of strongest postconditions and weakest preconditions.

From a technical point of view, the LCFG of a program can be constructed in three steps: first building the unlabeled CFG, from the syntax tree of the program; then assigning the first component of the labels by traversing the graph from *START* to *END*; and finally assigning the second component by traversing the graph in the reverse direction. In each of these traversals the label of each edge can be calculated *locally* from the labels of the (one or two) previous edges. In particular, for $1 \leq k \leq n$ we have $\overline{\text{spost}}_k(S, P) = \text{spost}(C_k, \overline{\text{spost}}_{k-1}(S, P))$ and $\overline{\text{wprec}}_k(S, Q) = \text{wprec}(C_k, \overline{\text{wprec}}_{k+1}(S, Q))$.

To illustrate the process of an interactive verification, please recall the program presented in the beginning of Chapter 6 used to calculate the income taxes in the United Kingdom. This program, written now using Java/JML notation, is shown again in Listing 9.2, considering as precondition $P = \text{age} \geq 18$ and as postcondition $Q = \text{personal} > 5750$.

If we verify the correctness of this program using the traditional approach, the VCGen module of GamaSlicer will produce a single formula to be sent to the prover and the result will be that the program is incorrect (Figure 9.7). Using this traditional approach, to understand which statements are leading to the incorrectness, the user must debug the program

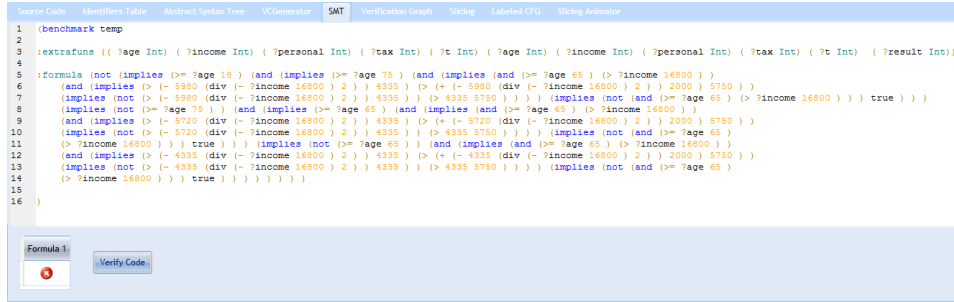


Figure 9.7: SMT code for the verification conditions of TaxesCalculation

manually. However, this is not the desirable way. Although the considered program is of small size, when dealing with larger programs this process becomes hard, boring and error prone. An interactive verification may help us to find the statements that prevent the program from being correct.

Let us start by visualizing the verification graph for the program under consideration. The result is the one in Figure 9.8. Now let us zoom-in on the subgraph corresponding to the second conditional statement in the method, and select the last edge inside the *then* branch. The edge condition sent to the prover is:

$$\exists v0 : age \geq 65 \ \&\& \ income > 16800 \ \&\& \ t > 4335 \\ \&\& \ personal == t + 2000 \rightarrow personal > 5750$$

which the prover identifies as being valid.⁴ Thus, the edges inside the branch and the assignment statement are shown in green and the *if/fi* nodes remain black (Figure 9.9), accordingly to the coloring rules in Definition 77 on page 197. This means that the statements inside the *then* branch are not the cause of failure. If we now pick the last edge inside the *else* branch, the following condition is sent to the prover:

$$\exists v0 : age \geq 65 \ \&\& \ income > 16800 \ \&\& \ t > 4335 \\ \&\& \ personal \neq t + 2000 \rightarrow personal > 5750$$

The result returned by the prover is now that the condition is not valid. Thus, the edges inside the branch, the assignment node, and the *if/fi* nodes all become red (Figure 9.10 — recall that although Definition 77 does not include the red color, it is here considered for visualization purposes, to help on the identification of segments of the code where a problem exists. At this step, we learn that the statement in this path is causing the procedure to be incorrect.

In fact this is not the only problem in this procedure. Repeating this process for the inner conditional inside the first one, we can observe that

⁴In fact it is the *negation* of this formula that is sent to the SMT solver, which returns *unsat*, meaning that the original formula is valid.

the statement $personal = 5720$ is also preventing the program from being correct.

In order to assist the user, during the interactive verification of a program, several features are available in order to make this process more intuitive and user friendly.

Once an edge is chosen (by clicking on it), the condition sent to the prover is shown in a Verification list, at the bottom of the window (Figures 9.9 and 9.10). The user is free to expand/collapse this list. When clicking on a list item, the edges/nodes related with the condition are momentarily highlighted. Also, in order to keep track of the edge conditions previously considered, complementing the Verification list, an history of images (also expandable/collapsible) is shown on the right (Figures 9.9 and 9.10). Once the selected condition has been processed, and according to the result returned by the prover, the current edge is colored: *red* if the prover returned false, *green* in case of true and *yellow* when the returned value is unknown.

Nodes in the graph will also be displayed in color, as follows: a node is shown in green if all its incoming and outgoing edges are green; it is shown in red if at least one of its incoming and outgoing edges is red; it is shown in black otherwise (i.e. no red edges and at least one black edge). For instance when an if node is shown in green, this means that the edge condition of the incoming edge is valid, i.e. the precondition of the corresponding conditional command poses no problems.

This interactive approach is thus useful when a program is incorrect and we want to detect which part of the program is causing such problems. In the traditional approach it could be hard to come up with the same conclusion just observing the generated verification conditions.

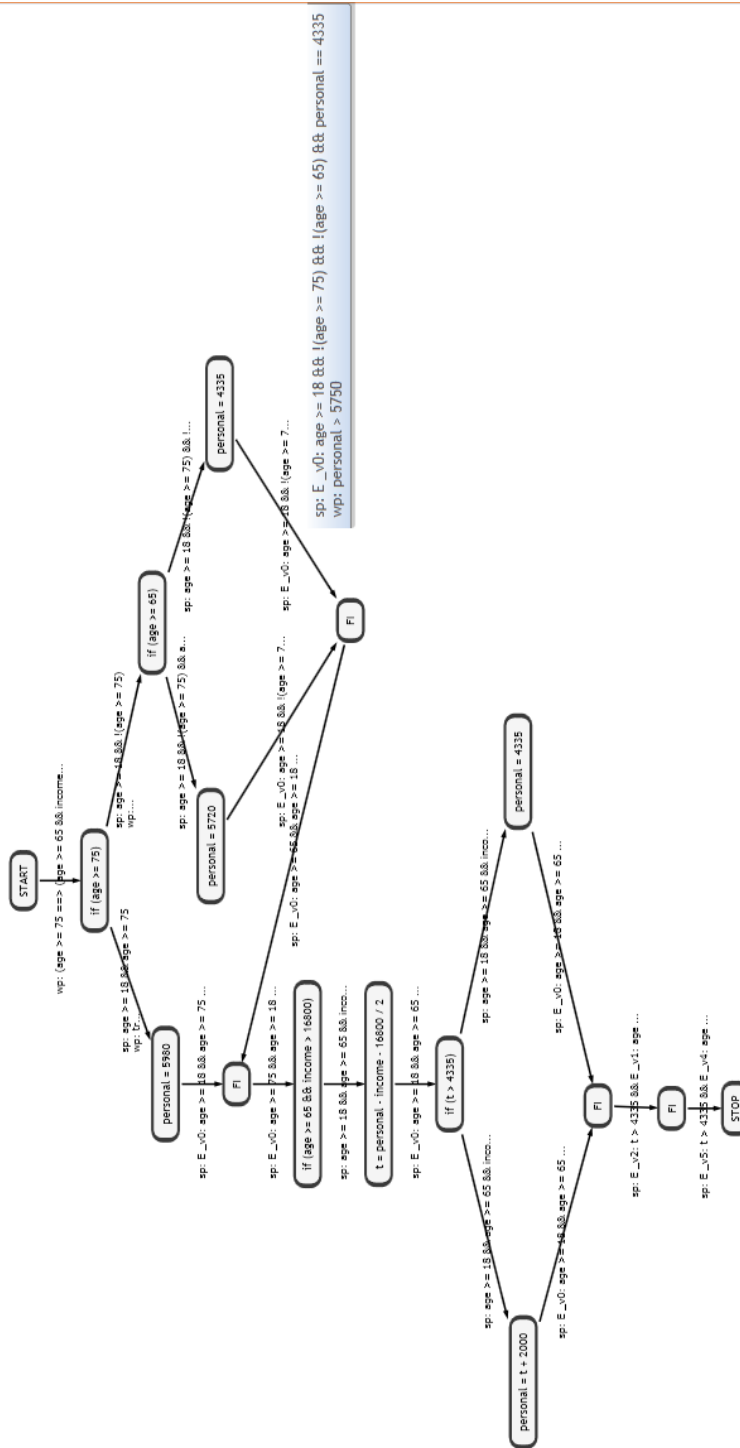


Figure 9.8: Verification Graph for the TaxesCalculation program

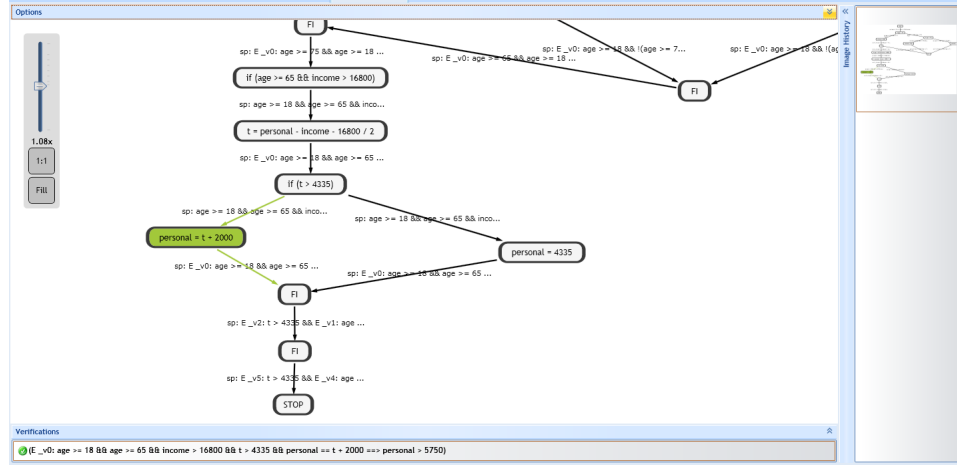


Figure 9.9: Verifying the Edge Conditions of Verification Graph for the TaxesCalculation program (1)

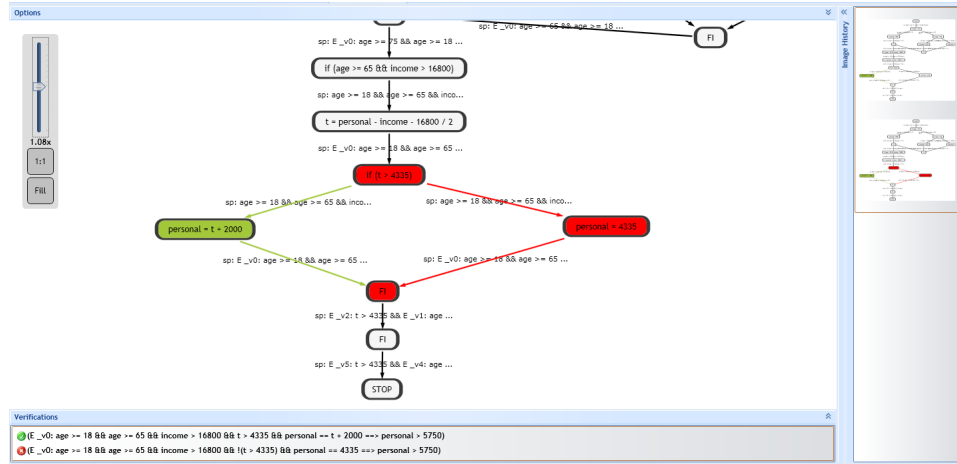


Figure 9.10: Verifying the Edge Conditions of Verification Graph for the TaxesCalculation program (2)

9.3 Slicing and Visualizing/Animating

The first aim that led to the design and implementation of GamaSlicer was precisely to provide a tool for the visualization of annotated programs and their semantic slices, as well as, for the animation of the slicing process — the lack of a similar tool (helpful for maintenance and teaching) was detected while reviewing the state of the art (Chapters 2 and 4). This section explains the contribution resulting from the present PhD work to this aim.

9.3.1 Assertion-based Slicing

GamaSlicer implements the precondition-, postcondition- and specification-based slicing algorithms (the original version and the new one). For the latter, it also provides an animator enabling the user to see the algorithm performing in a step-by-step mode through the animation of the LCFG.

To illustrate how this works on GamaSlicer, let us start with precondition-based slicing (postcondition-based works in a similar way).

Please recall the program used in Chapter 7, now written in Java/JML notation in Listing 9.3.

After submitting and parsing this source code, we can start by the slicing operation, choosing the precondition-based slicing algorithm. As expected, the statements in lines 18–20 were sliced off since the precondition $P = y > 10$ makes the statements in the **else** branch useless. The result exhibited in GamaSlicer is depicted in Figure 9.11.

Now suppose that we are interested in using both the precondition and the postcondition to slice the program, i.e., we intend to apply a specification-based slicing. This time, the resulting program is even smaller, since the statements in lines 14 and 15 are also sliced off (after line 13 the postcondition is met and thus all the statements after can be deleted). The result produced by GamaSlicer is depicted in Figure 9.12.

Consider now the annotated component in Listing 9.4 (for the sake of simplicity, a small procedure was deliberately selected).

Computing the weakest precondition and the strongest postcondition pair for each statement in the procedure 9.4, we obtain:

$$\begin{array}{ll}
 \overline{spost}_0 = x \geq 0 & \overline{wprec}_1 = x > 0 \\
 \overline{spost}_1 = \exists v. v \geq 0 \wedge x = v + 100 & \mathbf{x} = \mathbf{x} + \mathbf{100} \quad \overline{wprec}_2 = x > 100 \\
 \overline{spost}_2 = \exists v. v \geq 0 \wedge x = v - 200 & \mathbf{x} = \mathbf{x} - \mathbf{200} \quad \overline{wprec}_3 = x > 300 \\
 \overline{spost}_3 = \exists v. v \geq 0 \wedge x = v + 100 & \mathbf{x} = \mathbf{x} + \mathbf{200} \quad \overline{wprec}_4 = x > 100
 \end{array}$$

(notice that $\overline{sp}_0 \equiv P$ and $\overline{wp}_4 \equiv Q$)

The specification-based slicing algorithm looks for valid implications among the \overline{sp}_i and the \overline{wp}_j , for $0 \leq i \leq 3$ and $1 \leq j \leq 4$. For that, it

```

1 public class Ifslicing
2 {
3     int x, y;
4     public Ifslicing() {}
5
6     /*@ requires y > 10;
7       @ ensures x >= 0;
8       @*/
9     public void testIF()
10    {
11        if (y > 0)
12        {
13            x = 100;
14            x = x + 50;
15            x = x - 100;
16        }
17        else {
18            x = x - 150;
19            x = x - 100;
20            x = x + 100;
21        }
22    }
23 }

```

Listing 9.3: Example for precondition and specification-based slicing

```

1 /*@ requires x >= 0;
2   @ ensures x > 100;
3   @*/
4 public int changeX() {
5     x = x + 100;
6     x = x - 200;
7     x = x + 200;
8 }

```

Listing 9.4: Simple sequence of assignments

will test the implications: $\overline{sp}_0 \rightarrow \overline{wp}_2$, $\overline{sp}_1 \rightarrow \overline{wp}_3$, $\overline{sp}_0 \rightarrow \overline{wp}_4$, $\overline{sp}_1 \rightarrow \overline{wp}_3$, \dots , $\overline{sp}_2 \rightarrow \overline{wp}_4$.

For a larger procedure, it becomes harder to identify this set of implications and specially difficult to understand which statements can be eliminated (remember that a statement sequence can be discarded whenever an implication is true). In this context, the main idea behind the animator of GamaSlicer came up: to use the same LCFG used in the interactive verification to display the implications and animate the verification process.

Submitting the program to GamaSlicer and starting by the animation of the specification-based slicing algorithm, its initial LCFG will be displayed — the layout of this graph corresponds to a global graph layout (suitable for all the animation process) and will be the one used during all the slicing phases, avoiding extra effort to interpret changes in the nodes localization.

Graph animation, which consists in showing a sequence of n graphs, is not a novelty but it still presents some interesting challenges [DG02]. The naive approach is to re-calculate the whole layout in each update. However, this can lead to some confusion as it does not preserve the *mental map* of the user who is following the animation. Mental map refers to the abstract structural information that a user forms when looking at the layout of a graph, which facilitates the navigation in the graph and its comparison with

other graphs. An obvious challenge is *how to display each graph in the sequence while preserving the basic layout*. In GamaSlicer it was adopted a solution to this problem known as *foresighted layout* as proposed in [DGK00] (also referred in subsection 5.3.3 of Chapter 5). Given a sequence of n graphs, a global layout is computable, suitable for each one of the n graphs. In the simplest case, this global layout can match the super-graph of all graphs in the sequence.

Once the global layout is computed and displayed, it is possible to step forward. The arrow that corresponds to the implication being tested is highlighted — starting in black and finishing in a color that depends on the proof result — see Figure 9.13. The proof obligation being tested is also shown. If the implication evaluates to true, then the (green) arrow is added to the graph. At the end of the step, the proof is added to the textual history (displayed on bottom) and the complete snapshot of the graph is added to the image history (displayed on the upper-right corner).

A backward step allows the user to recover the animation track and gives a clear perception between the previous and current state.

After all slicing steps (all steps are depicted in Figures 9.13 to 9.19), the shortest-path among all the green paths from Start to Stop nodes is computed and displayed. The statements to be sliced off are then highlighted. See Figure 9.20.

The animation process ends up displaying the final slice graph that describes the program obtained, as shown in Figure 9.21.

Through this process, the user can check which implications led to the sliced statements, and thus understand the final result.

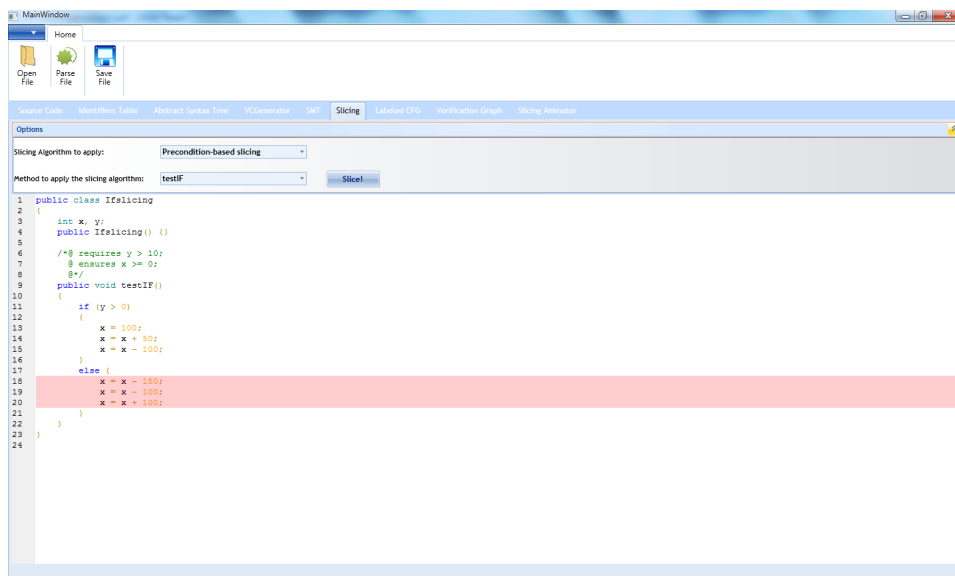


Figure 9.11: Precondition-based slicing applied to program in Listing 9.3

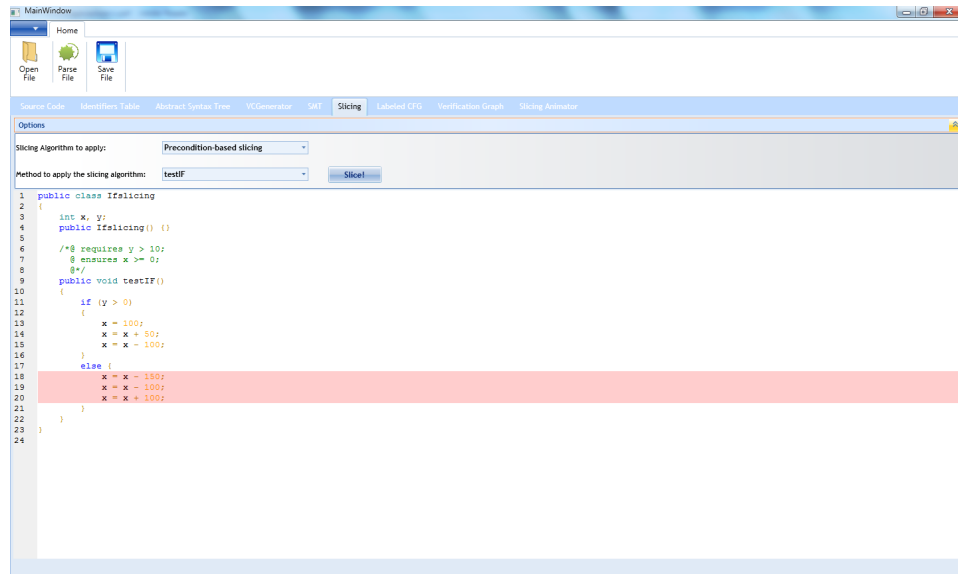


Figure 9.12: Specification-based slicing applied to program in Listing 9.3

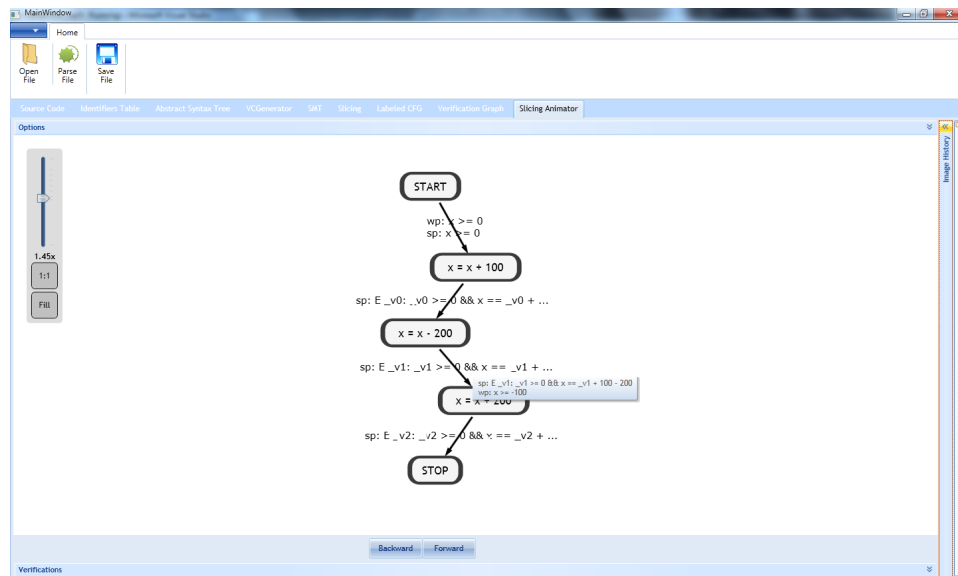


Figure 9.13: Animating specification-based slicing applied to program in Listing 9.4 (Step 1)

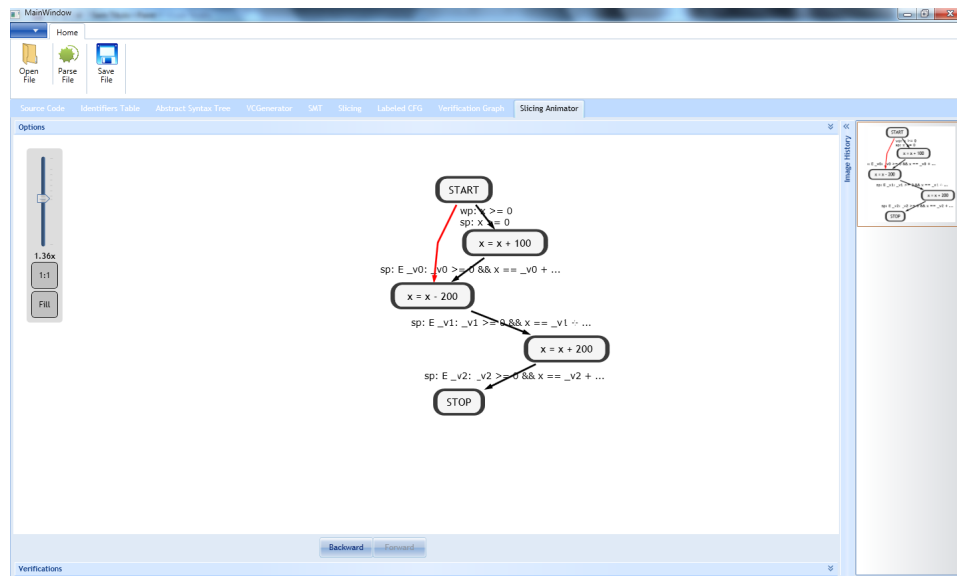


Figure 9.14: Animating specification-based slicing applied to program in Listing 9.4 (Step 2)

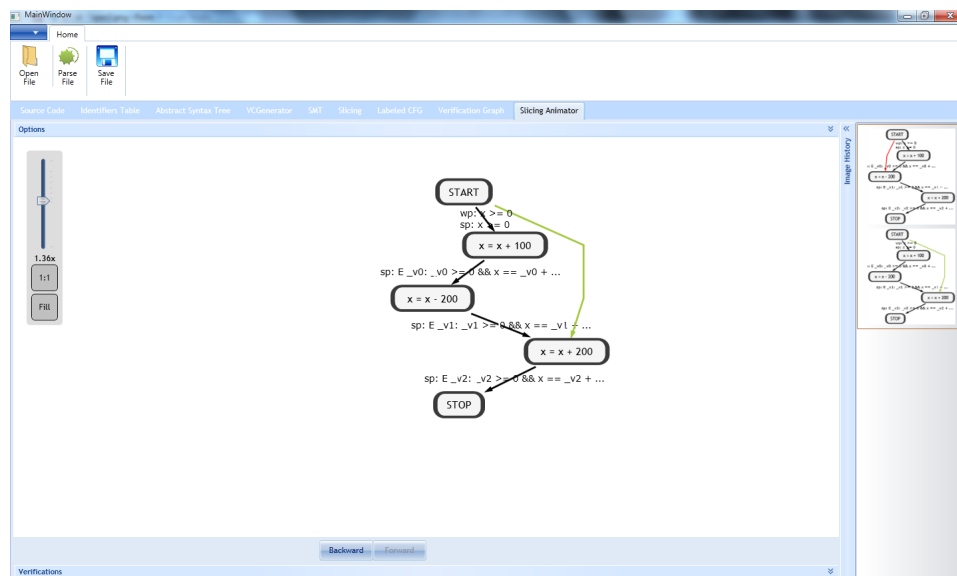


Figure 9.15: Animating specification-based slicing applied to program in Listing 9.4 (Step 3)

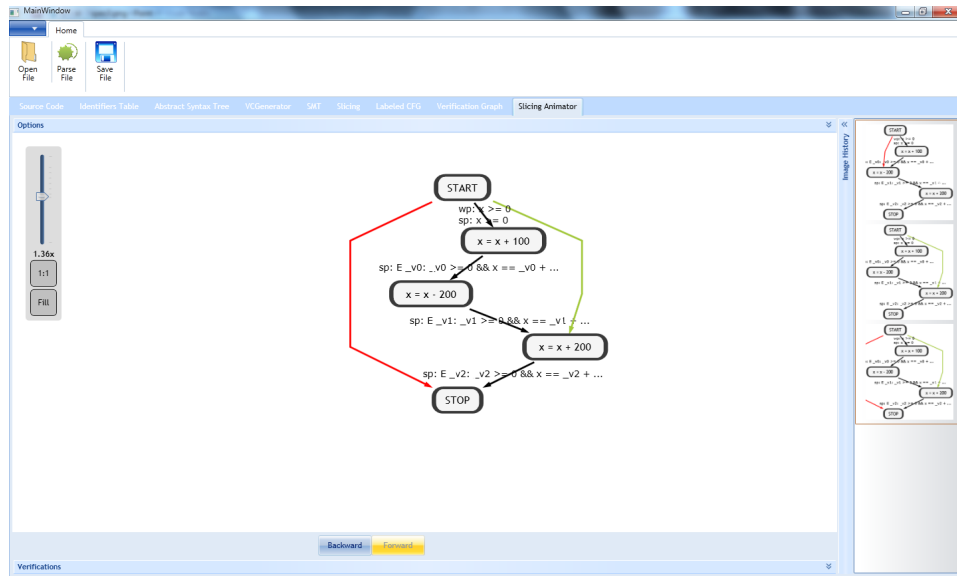


Figure 9.16: Animating specification-based slicing applied to program in Listing 9.4 (Step 4)

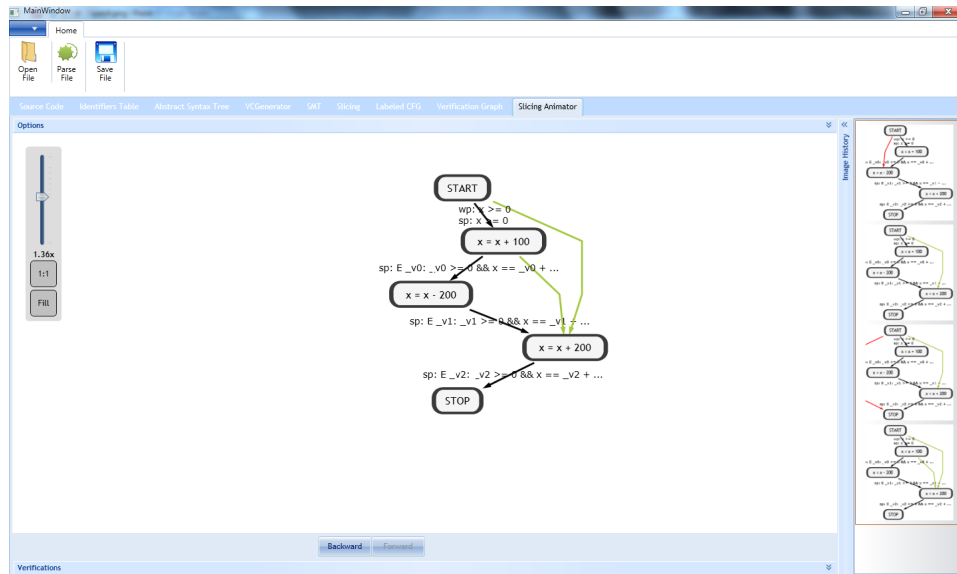


Figure 9.17: Animating specification-based slicing applied to program in Listing 9.4 (Step 5)

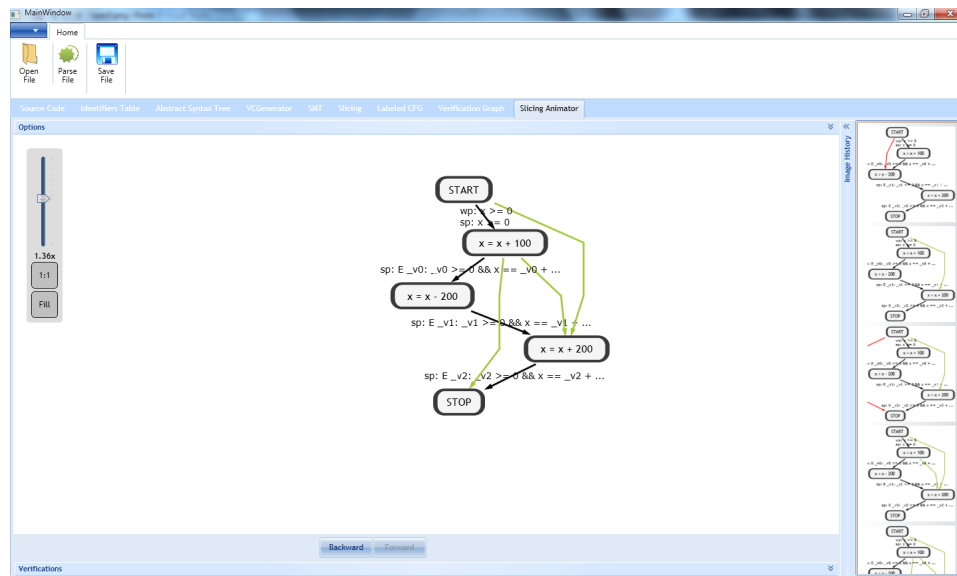


Figure 9.18: Animating specification-based slicing applied to program in Listing 9.4 (Step 6)

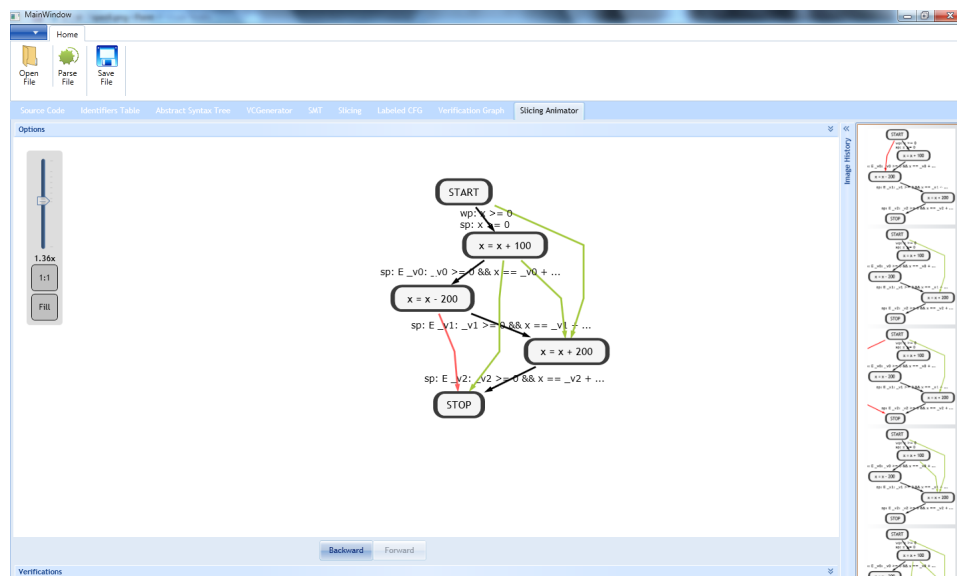


Figure 9.19: Animating specification-based slicing applied to program in Listing 9.4 (Step 7)

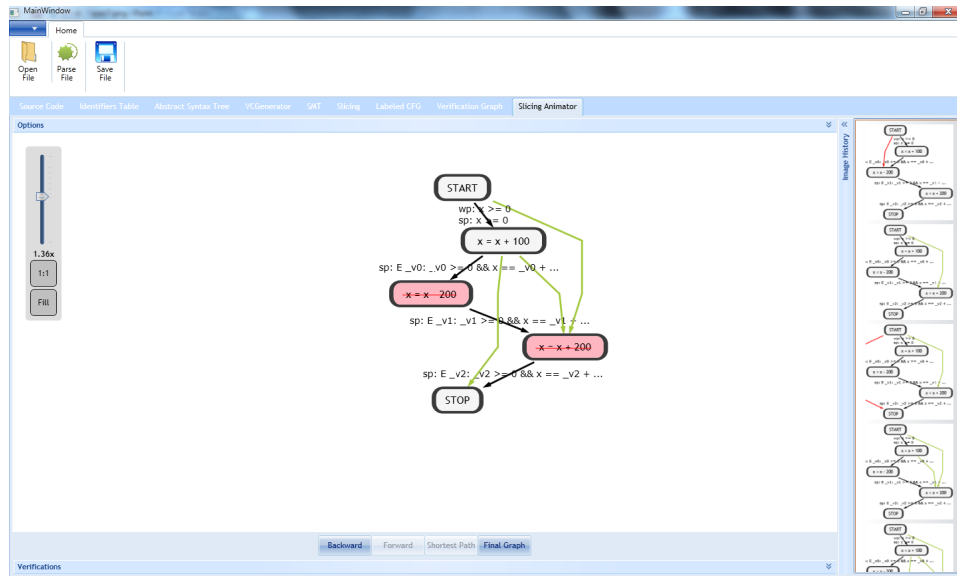


Figure 9.20: Animating specification-based slicing applied to program in Listing 9.4 (Step 8 — Shortest Path)

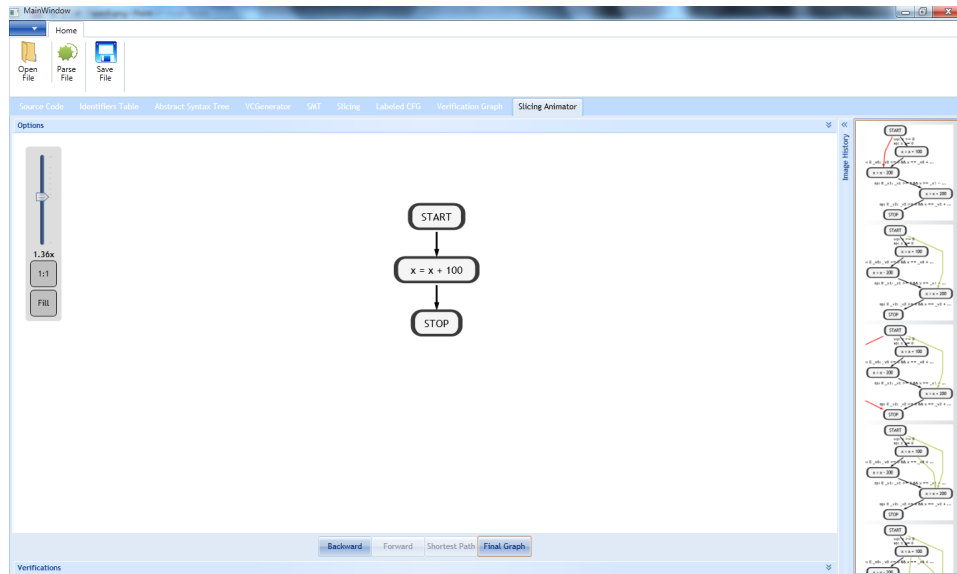


Figure 9.21: Animating specification-based slicing applied to program in Listing 9.4 (Step 9 — Final Slice Graph)

9.3.2 Contract-based Slicing

In this section it is illustrated the concept of *contract-based slice* introduced in Chapter 8.

Consider as an example an extract from a class Π in Listing 9.5 containing an annotated method, called **OpersArrays**, which computes several array operations: the *sum of all elements*, the *sum of all the even elements*, the *iterated product*, and the *maximum* and *minimum*. Π contains two other methods **Average** and **Multiply**; the former computes the average of the elements belonging to the array, and the latter multiplies the product of the array by a parameter y . The code contains appropriate contracts for all methods, as well as loop invariants. All 3 methods are correct with respect to their contracts – we assume this has been established beforehand.

Moreover suppose that **OpersArrays** is a private method, i.e. in a given context it is known that it will not be invoked externally. As can be observed in Listing 9.5: method **Average** calls the method **OpersArrays** and then uses the calculated value of the variable **summ**; and **Multiply** calls **OpersArrays** and then uses the value of variable **productum**.

In this context, it makes sense to calculate a contract-based slice of this class with $\mathcal{S} = \{\text{Average}, \text{Multiply}\}$, which will result in the method **OpersArrays** being stripped of the irrelevant code.

In order to calculate $\text{progslice}_{\mathcal{S}}(\Pi)$, T_{post} is first initialized as

$$T_{\text{post}}[\text{OpersArrays}] := \top$$

After performing the first step of the algorithm we set

$$T_{\text{post}}[\text{OpersArrays}] := \top \ \&\& \ Q_1 \ \&\& \ Q_2$$

where (calculations omitted)

$$Q_1 = \frac{\text{summ}}{\text{length}} == \frac{\text{sum}\{\text{int } i \text{ in } (0 : \text{length}); a[i]\}}{\text{length}}$$

$$Q_2 = 0 \leq i \leq y \ \&\& \ q == i \times \text{productum}$$

So, as the component **OpersArrays** is called twice in the context of the two previous components, the result is the weaker postcondition

$$\top \ \&\& \ Q_1 \ \&\& \ Q_2.$$

The final step in the calculation of the slice using table T gives us

$$\text{progslice}_{\mathcal{C}}(\Pi) \doteq \{\text{progslice}_{\circ}(\text{Average}), \text{progslice}_{\circ}(\text{Multiply}), \text{progslice}_{\mathcal{S}}(\text{OpersArrays})\}$$

```

1  int summ, sumEven, productum, maximum, minimum;
2  int[] a = new int[100];
3  int length = 100;
4
5  /*@ ensures \result == (\sum int i; 0 <= i \&\& i < length; a[i])/length;
6  @*/
7  public double Average()
8  {
9      OpersArrays();
10     double average = summ / length;
11     return average;
12 }
13
14 /*@ requires y >= 0;
15     ensures \result == (\product int i; 0 <= i \&\& i < length; a[i])*y;
16 @*/
17 public int Multiply(int y)
18 {
19     int q = 0, i = 0;
20     OpersArrays();
21     x = productum;
22     /*@ loop_invariant (0 <= i) \&\& (i <= y) \&\& (q == (i*x));
23     @*/
24     while (i < y)
25     {
26         q = q + x;
27         i = i + 1;
28     }
29     return q;
30 }
31
32 /*@ ensures (summ == (\sum int i; 0 <= i \&\& i < length; a[i])) \&\&
33     (productum == (\product int i; 0 <= i \&\& i < length; a[i])) \&\&
34     (sumEven == (\sum int i; 0 <= i \&\& i < length;
35         ((a[i] % 2) == 0 ? a[i] : 0))) \&\&
36     (maximum == (\max int i; 0 <= i \&\& i < length; a[i])) \&\&
37     (minimum == (\min int i; 0 <= i \&\& i < length; a[i]));
38 @*/
39 private void OpersArrays()
40 {
41     maximum = a[0];
42     minimum = a[0];
43     summ = 0;
44     sumEven = 0;
45     productum = 1;
46     int n = 0;
47
48     /*@ loop_invariant (n <= length) \&\&
49         (summ == (\sum int i; 0 <= i \&\& i < n; a[i])) \&\&
50         (productum == (\product int i; 0 <= i \&\& i < n; a[i])) \&\&
51         (sumEven == (\sum int i; 0 <= i \&\& i < n;
52             ((a[i] % 2) == 0 ? a[i] : 0))) \&\&
53         (maximum == (\max int i; 0 <= i \&\& i < n; a[i])) \&\&
54         (minimum == (\min int i; 0 <= i \&\& i < n; a[i]));
55 @*/
56     while(n < length)
57     {
58         summ = summ + a[n];
59         productum = productum * a[n];
60         if ((a[n] % 2) == 0) { sumEven = sumEven + a[n]; }
61         if (a[n] > maximum) { maximum = a[n]; }
62         if (a[n] < minimum) { minimum = a[n]; }
63
64         n=n+1;
65     }
66 }

```

Listing 9.5: Annotated method OpersArrays

```

1  public void OpersArrays()
2      /*@ ensures (summ == (\sum int i; 0 <= i && i < length; a[i])) &&
3          (productum == (\product int i; 0 <= i && i < length; a[i]))
4          @*/
5      {
6          summ = 0;
7          productum = 1;
8          n = 0;
9          /*@ loop_invariant (n <= length) &&
10             (summ == (\sum int i; 0 <= i && i < n; a[i])) &&
11             (productum == (\product int i; 0 <= i && i < n; a[i]))
12             @*/
13         while(n < length)
14         {
15             summ += a[n];
16             productum = productum * a[n];
17             n=n+1;
18         }
19     }
20 }

```

Listing 9.6: Sliced method `OperArrays`

Calculating

`slice(body(OperArrays), pre(OperArrays), $T_{post}[\text{OperArrays}]$)`

results in cutting the statements present in lines 40, 41, 43, and 59-61 after removing from the invariant the predicates in lines 50-53, which contain only occurrences of variables that do not occur in $T_{post}[\text{OperArrays}]$.

The final sliced method is shown in Listing 9.6. The other two methods remain unchanged.

GamaSlicer also implements this algorithm in an interactive mode. At the beginning, after the selection of this functionality, the initial postconditions table is set (see Figure 9.22). During the first step (the calculation of the verification conditions), each time a function call is found, the respective line in the source code as well as the entry of the procedure in the table are highlighted (see Figures 9.23 and 9.24). At the end, when no more calls are found, the user selects the application of the contract-based slicing algorithm to the original program using the new postconditions (step 2). As result, the lines to be removed from the original program are highlighted. The final program is shown in Figure 9.25.

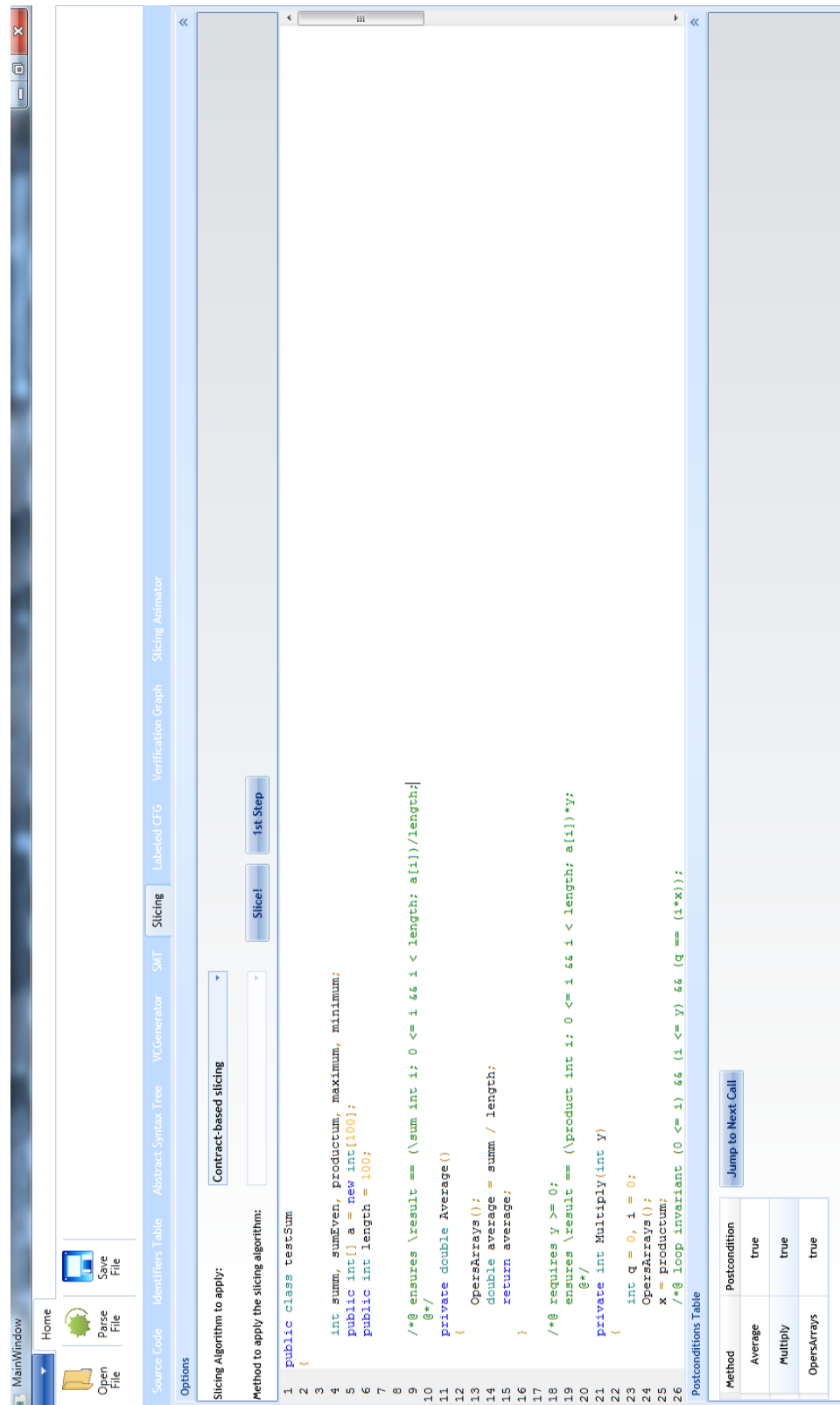


Figure 9.22: The initial postconditions table

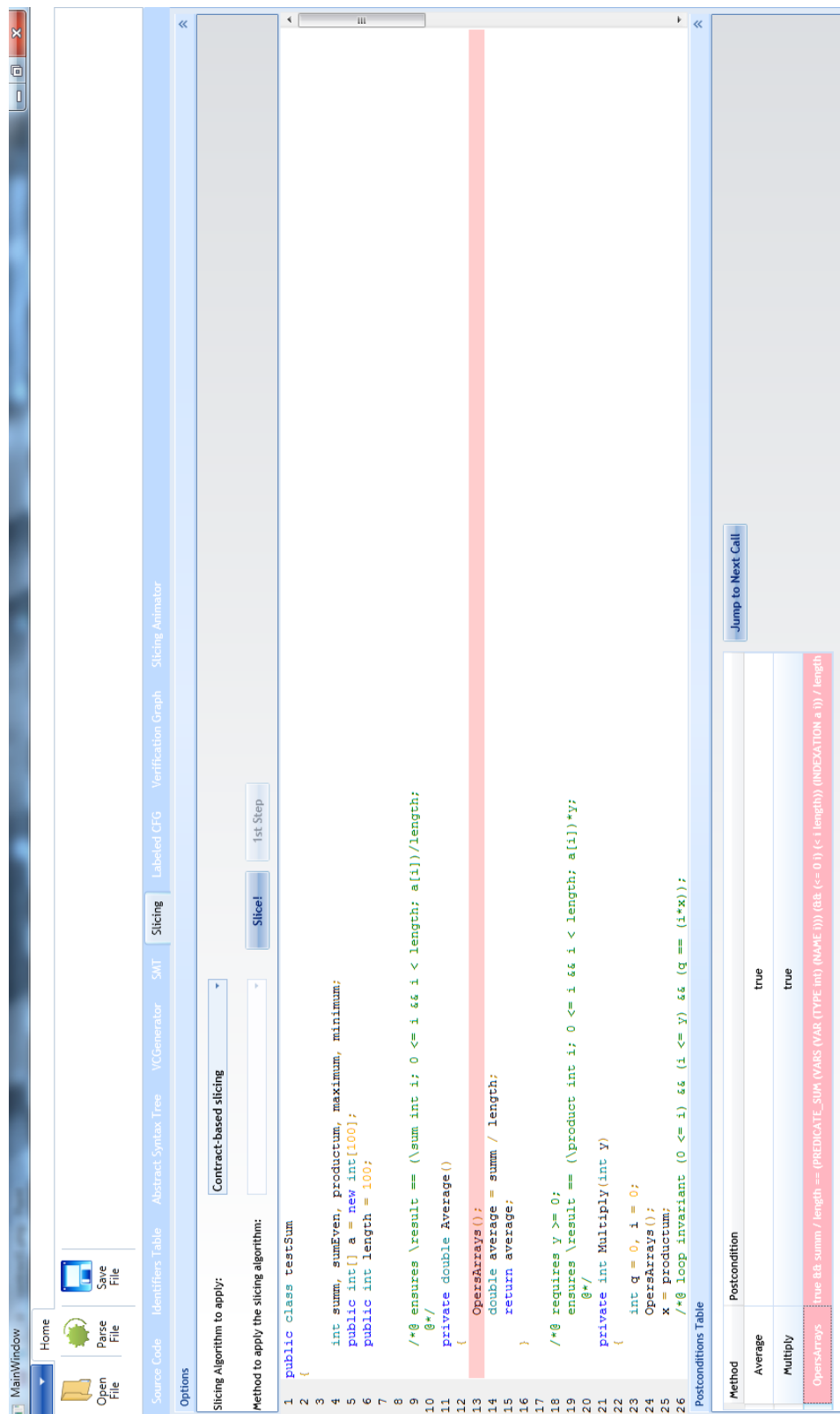


Figure 9.23: Changing postconditions table when a function call is found (1)

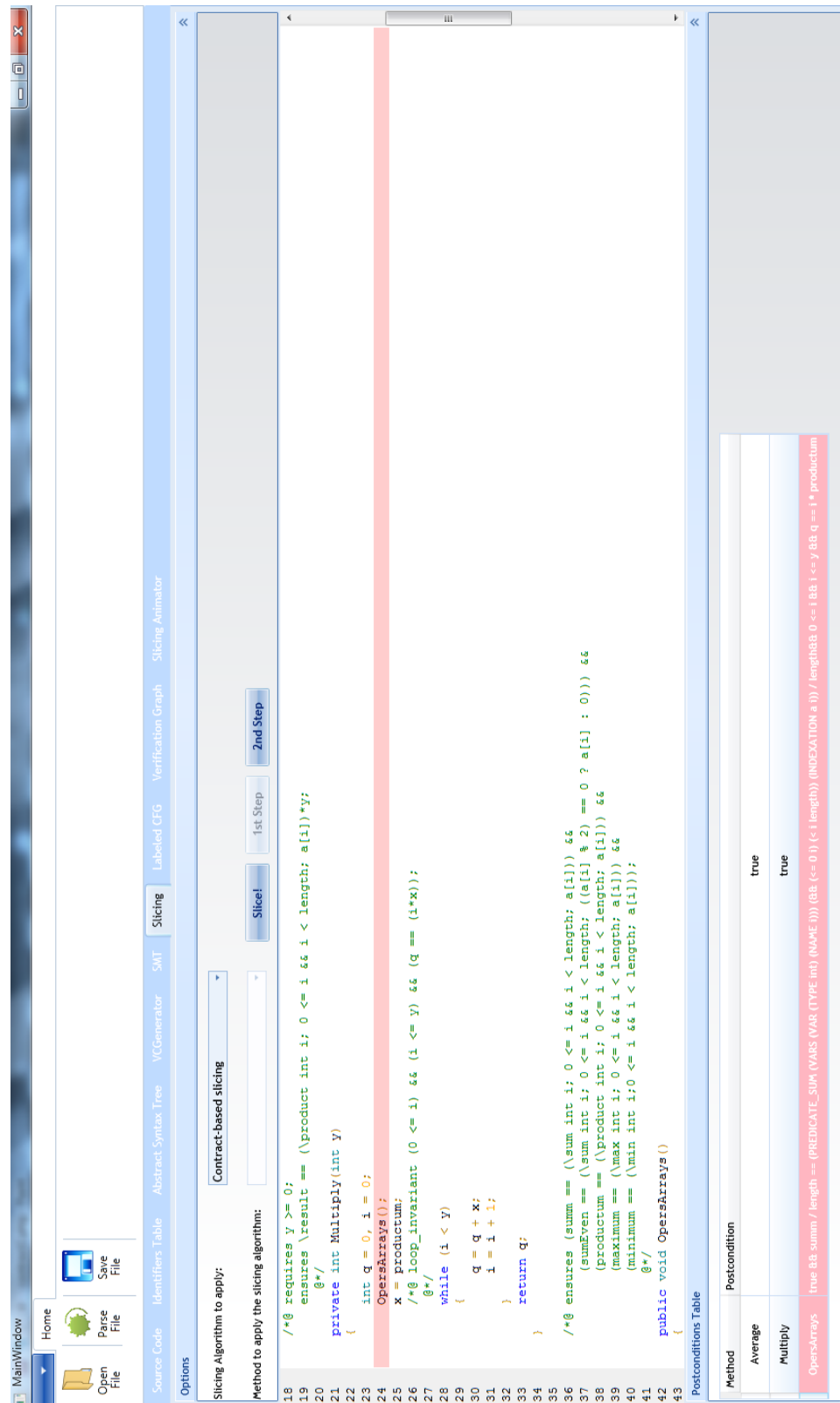


Figure 9.24: Changing postconditions table when a function call is found
(2)

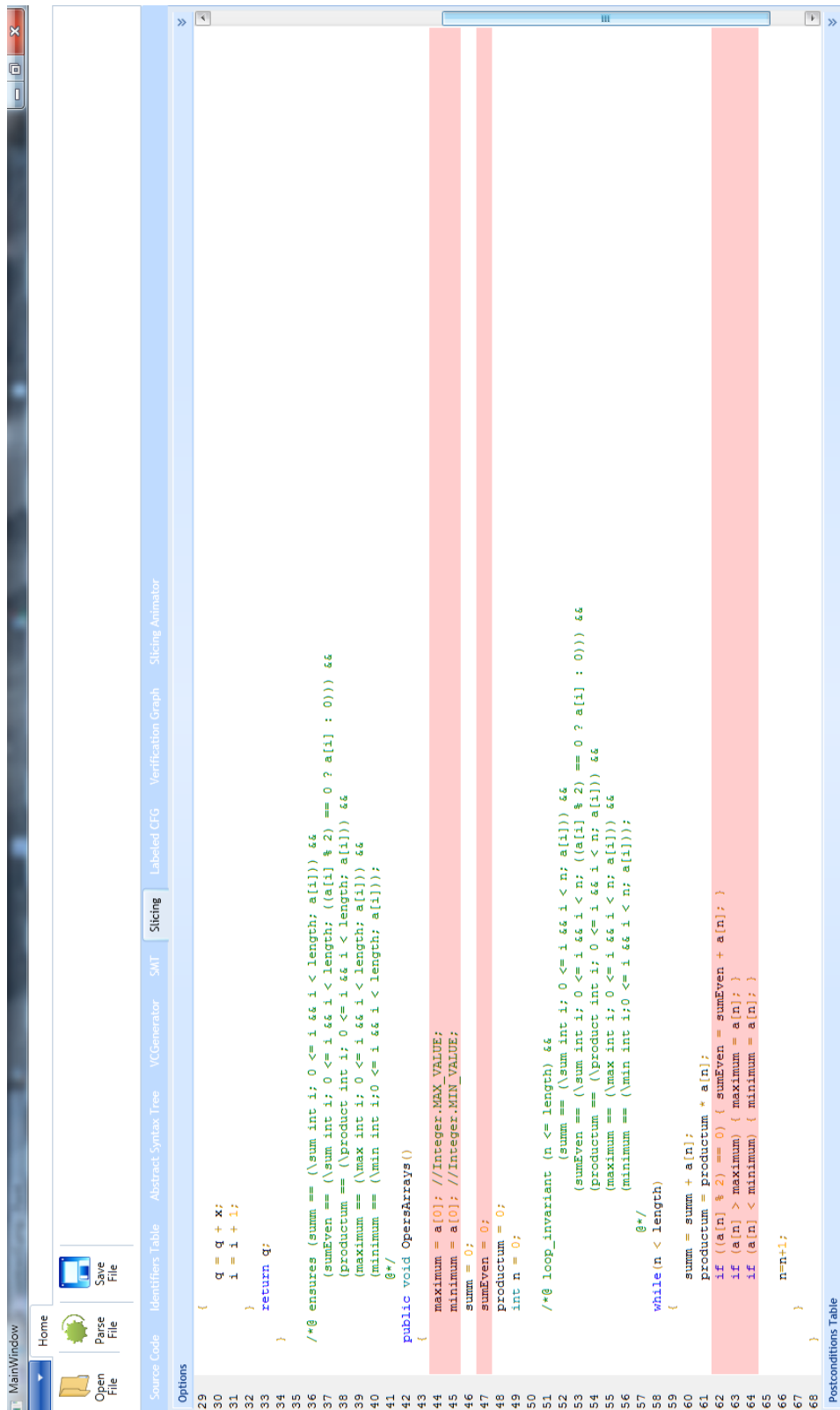


Figure 9.25: Final program with the lines to be removed highlighted

Chapter 10

Conclusion

I love to travel, but I hate to arrive.

Albert Einstein, 1879 — 1955

This thesis delves into the problematics of slicing programs based on their semantics rather than in their syntax. This idea was presented in Chapter 7. It was shown how the forward propagation of preconditions and the backward propagation of postconditions can be combined in a new slicing algorithm that is more precise than the existing specification-based algorithms. The algorithm is based on (i) a precise test for removable statements, and (ii) the construction of a *slice graph*, a program control flow graph extended with semantic labels and additional edges that “short-circuit” removable commands. It improves on previous approaches in two aspects: it does not fail to identify removable commands; and it produces the smallest possible slice that can be obtained (in a sense that was made precise). Iteration is handled through the use of loop invariants and variants to ensure termination. Applications of these forms of slicing were also discussed, including the elimination of (conditionally) unreachable and dead code. A comparison with other related notions was also given.

The notion of assertion-based slicing was then generalized to programs with multiple procedures under the concept of *contract-based slicing* in Chapter 8 — both *open* and *closed* versions were proposed as a way to remove code unnecessary to the correctness of a given program. The motivation was to bring to the inter-procedural level the power of assertion-based slicing, which we believe has great application potential and will surely become more popular in coming years, profiting from advances in verification and automated proof technology.

A master thesis, entitled “*Contracts and Slicing for Safety Reuse*”, was developed in close connection with the work reported in this document, but it explored a different facet of slicing programs based on their con-

tracts: it focuses on the verification of every call to an annotated component, checking if the actual calling context preserves the callee precondition. The Caller-based Slicing algorithm and examples can be seen in more detail in [Are10, AdCHP11]. Currently, in order to consider a more realistic language and produce more accurate results, JavaPathFinder Symbolic Executor [BHPV00] is being integrated in the tool (see also subsection 3.5.3 for more details about JavaPathFinder).

The thesis also includes contributions in the area of Program verification. As discussed, the *verification problem* considered here could be stated as the problem of determining whether a program Π meets its specification. If Π is a large program, it can be decomposed into smaller pieces (procedures), each of which will be annotated with a local specification pair (P', Q') , also called a *contract* (its own entry and exit invariants). The procedures will be verified separately, and Π will then be considered correct if all their procedures are correct with respect to their contracts.

However, when dealing with an incorrect program, it is difficult to find which statements are causing the incorrectness through the traditional verification approach. Chapter 6 introduced an application of labeled control flow graphs as verification graphs, a structure to organize a set of verification conditions in which the propagation of logical information forward and backward is interactively commanded by the user. By visualizing the labeled control flow graph of a procedure, the user can select the edge conditions to be sent to the prover, and thus keep track of valid and invalid conditions. This way, the user can easily identify the erroneous paths.

The main applications of LCFGs in this thesis were thus for: interactive verification as shown in Chapter 6, giving the user the power to guide the generation of VCs; and for slicing as shown in Chapter 7. We remark however that verification graphs contain all the different sets of VCs that can be obtained for a block of code combining forward and backward propagation, and moreover the VCs are organized in a way that facilitates the association with execution paths: an edge condition is valid if and only if no execution path containing that edge produces an error. These are interesting properties in themselves, and we believe that in addition to interactive verification, LCFGs can be used as the basis for defining VCGens based on different strategies, combining forward and backward propagation of assertions. This is a topic that has not been explored in the literature, but which seems to combine the best of the two standard approaches to the generation of verification conditions.

In order to prove the validity of all these ideas, the GamaSlicer tool was developed. It implements all the assertion- and contract-based slicing algorithms as well as a visualizer/animator for the slicing algorithms and for the interactive VCGen.

GamaSlicer can be a useful teaching tool, since it allows students to observe step-by-step how verification conditions are generated. For example, since loop invariants and procedure contracts are annotated into the code, the only arbitrary choice is in the rule for the sequence command $C_1 ; C_2$, in which an intermediate assertion R must be guessed. This is the motivation for introducing a *strategy for the construction of proof trees*, based on the notion of weakest precondition. This gives a mechanical method for constructing derivations, which can alternatively be written as a VCGen.

GamaSlicer can also be used to help in detecting and fixing errors, both in annotations and in code, since it exhibits in a versatile and user-friendly tree fashion all the rules used to generate the proof obligations; this output can optionally be obtained by executing the VCGen algorithm step-by-step. The proof obligations, written in the SMT-Lib language, are also displayed, allowing the user to analyze the formulae and their verification status.

Another contribution of GamaSlicer is for program comprehension: slicing a method according to different contracts (by weakening the original contract) allows one to understand the relation between each code section and each part of the initial contract.

An extension to GamaSlicer, GamaPolar, was developed under the context of the master thesis mentioned above. It implements a new slicing algorithm called *caller-based slicing* that basically consists in a two-pass algorithm where: the first step is the implementation of the traditional backward slicing with a slicing criterion concerning the call under consideration, and its arguments (the ones that are also present on the callee precondition); and the second step is to verify if the statements that result from the first step will respect the contract with the callee. The tool produces a set of errors and warnings about the (possible) violations with respect to the precondition under consideration.

After summarizing the main novelties of the PhD work here reported, some uses that evidence their contribution will be described.

Applications. The applications of both assertion-based slicing and contract-based slicing are multiple.

In particular, with respect to postcondition-based slicing, Comuzzi and Hart, in their paper, give a number of examples of its usefulness, based on their experience as software developers and maintainers. Their emphasis is on applying slicing to relatively small fragments of big programs, using postconditions corresponding to properties that should be *preserved* by these fragments. Suppose one suspects that a problem was caused by some property Q being false at line k of a program S with n lines of code. We can take the subprogram S_k consisting of the first k lines of S and slice it with respect to the postcondition Q . This may result in a suffix of S_k being sliced

off, say from lines i to k , which means that in order for Q to hold at line k , it must also hold at line i . The resulting slice is where the software engineers should now concentrate in order to find the problem (a similar reasoning applies if the sequence of lines removed is not a suffix of S_k).

A related situation occurs when the property must deliberately be violated in some part of the code. This is typical for instance of code running as a thread of a concurrent program, with Q being true outside a critical section executed by the thread at some point, and false inside that section. Q is true before entering the critical section and will be true after leaving it, so postcondition-based slicing can be used to study the correct behavior of the code with respect to that section. Similarly, the property may correspond to some invariant of a data structure, say a balanced binary search tree that will temporarily be unbalanced (or even inconsistent) while a new element is being inserted.

Safety properties may also be studied in this way. Examples include for instance

- array accesses $u[e]$, with safety property “the value of expression e stands between 0 and $N - 1$ ”, with N the allocated size of the array;
- pointer dereferencing accesses $*p$ with safety property “ p points to a properly allocated memory region”;
- procedure invocations, with safety property “the precondition of the invoked procedure is satisfied”.

With respect to specification-based slicing, it may be useful to apply it to code already annotated with specifications, following the principles of *design by contract*. For code that has been developed in this way, it is cheap to apply specification-based slicing techniques, based on the specification information that is already present in the code.

A first application in this context is the elimination of unnecessary code. A piece of software that has already been proven correct with respect to a specification may well contain code that is not actually playing any useful role regarding that specification. This unnecessary code that may have been introduced during development is not detected by the verification process itself, but slicing the program with respect to the proven specification will hopefully remove such code.

A different application is concerned with code that has been verified but is now being used in a specialized context, i.e. the specification that is required for a given use of the code is actually weaker (because a stronger precondition is present, and/or a weaker postcondition is required) than the proven specification. A typical situation is software reuse. For instance, consider a library containing a procedure that implements a traversal of some data structure, and collects a substantial amount of information in that

traversal. It may be the case that for a given project one wants to reuse this procedure without requiring all the information collected in the traversal. In this case the procedure will be invoked with a weaker specification, and it makes sense to produce a specialized version to be included in the current project. A specification-based slice can be computed to this effect.

Future work. With respect to future work, several ways can be pursued:

- It would be interesting to see how our work on verification graphs relates with the study of the effects of splitting verification conditions [LMS08]. The authors have shown that splitting VCs (i.e. having a bigger number of smaller conditions) can lead to substantial improvements with respect to the performance of the SMT prover, and also with respect to the quality of the error messages produced when verification fails. Verification graphs may offer a method for defining criteria for systematically splitting VCs.
- One very simple extension, which would increase even more the advantages of interactive VC generation, would be to give the user the possibility to insert assertions at arbitrary points of the verification graph. An intermediate assertion simultaneously creates a verification condition (the assertion must hold at the point where it is inserted), and provides additional context information that can be used in subsequent VCs. Technically it suffices to consider a new command **assert** ϕ , which has no operational effect, and:

$$\begin{array}{ll} \text{wprec}(\text{assert } \phi, Q) = \phi & \text{spost}(\text{assert } \phi, P) = \phi \\ \text{VC}^w(\text{assert } \phi, Q) = \{\phi \rightarrow Q\} & \text{VC}^s(\text{assert } \phi, P) = \{P \rightarrow \phi\} \end{array}$$

- Another interesting thing to add to verification graphs would be an online annotation editing facility. The programs considered along this document are annotated with contracts and loop invariants. The edge conditions are generated taking these annotations as inputs. It makes sense for an interactive tool to allow annotations to be modified online, which does not require parsing the program and reconstructing the verification graph. Only the modified annotation must be parsed, and then the edge conditions must be recalculated according to the verification strategy selected by the user.
- With respect to assertion-based and contract-based slicing algorithms, one obligatory step will be to calculate weakest preconditions using Flanagan and Saxe's algorithm [FS01], which avoids the potential exponential explosion in the size of the conditions generated, keeping the algorithm presented within quadratic time. Alternatives to strongest

postcondition calculations will also be explored, to eliminate the use of existential quantifiers. One such alternative is the notion of *update*, as used prominently in the dynamic logic of the KeY system [ABB⁺05].

- One main challenge is to produce a robust tool for intermediate code of a major verification platform, such as Why or Boogie. This would allow us to test the ideas with realistic code, since several verification tools for languages like C, Java, or C# are based on these platforms.

Currently, a master thesis, entitled “*GamaBoogie, a Contract-based Slicer for Boogie Programs*”, is being developed to explore such idea. A preliminary study of Boogie has been done resulting in a visualizer to help in the understanding of Boogie programs, and also the transformation that programs suffer when translating them from languages like C# or C to Boogie. The tool being developed is an extension to GamaSlicer and has the name of GamaBoogie.

I hope to have given enough arguments in favor of *the use of specifications as slicing criteria to obtain semantics-based slices*, and that this is a worthwhile field deserving more research.

Appendix A

Java Modeling Language

JML is a formal language that allows to specify both the syntactic interface of Java code (e.g. visibility and other modifiers, type checking information, etc) and its behavior (describes what should happen at runtime when the code is used).

A JML specification is written in special Java annotated comments, which start with an at-sign (@) and usually are written before the header of the method they specify. Thus, comments starting either with `/*@ ... @*/` or `//@` are considered JML annotations. Different from traditional comments, a formal specification in JML is machine checkable since these specifications represent boolean expressions (*assertions*). Checking the specification can help isolate errors before they propagate too far.

Similar to Eiffel [Mey87], the language that first came up with the Design-by-Contract approach, JML uses Java's expression to write the assertions, such as preconditions, postconditions and invariants. However, because these expressions lack some expressiveness to the writing of behavioral specifications, JML uses extended Java expressions. Some of these extensions are listed in Table A.1 and they include: a notation for describing the result of a method (`\result`), different types of implication, a way of referring to the pre-state value of an expression (`\old(.)`), several kinds of quantifiers (universal quantifier — `\forall`, existential quantifier — `\exists`, and generalized quantifiers — `\sum`, `\product`, `\min`, `\max`).

The quantifiers `\sum`, `\product`, `\max` and `\min` are quantifiers that return respectively the sum, product, maximum and minimum of their body expressions when the quantified variables satisfy the given range expressions. For example, an expression `(\sum int x; 1 <= x && x <= 6; x)` denotes the sum of values between 1 and 6 inclusive (i.e. 21).

JML uses a `requires` clause to specify the client's obligation, an `ensures` clause to specify the programmer's obligation; and a `loop_invariant` clause to specify the properties that need to be checked at each loop iteration. Other kind of clauses are available in JML (for instance, JML allows the

Syntax	Meaning
<code>\result</code>	result of method call
<code>a ==> b</code>	a implies b
<code>a <== b</code>	a follows from b (i.e. b implies a)
<code>a <==> b</code>	a if and only if b
<code>a <!=> b</code>	not (a if and only if b)
<code>\old(E)</code>	value of E in pre-state

Table A.1: Some of JML's extension to Java expressions

```

1  //@ requires x >= 0;
2  /*@ ensures \result >= \old(x);
3  @*/
4  public int Sum(int x) {
5      int i = 0, sum = 0;
6
7      /*@ loop_invariant 0 <= i && i <= x &&
8      @*/
9      while (i <= x) {
10         sum += i;
11     }
12     return sum;
13 }

```

Listing A.1: JML specification to describe the method's behavior

specification of what exceptions a method may throw) but they are left out of scope, since are not be used in the context of the work here reported. More information about JML can be found in [LC03] and [LBR06]. Listing A.1 shows an example of a JML specification describing the behavior of a method through the use of a precondition and a postcondition. Additionally, a loop invariant is added.

Figure A.1 shows a fragment of the Java grammar extended to deal with JML specifications.


```

MethodDeclaration → MethodSpecification Type Ident
                  FormalArgs CompoundStatement

MethodSpecification → RequiresClause* EnsuresClause*

RequiresClause → /*@ requires Expression ; @*/

EnsuresClause → /*@ ensures Expression ; @*/

CompoundStatement → { Statement+ }

Statement → CompoundStatement | ...
           | Maintaining* while ( Expression ) Statement

Maintaining → /*@ loop_invariant Expression ; @*/

Expression → ImpliesExpression
            | ImpliesExpression (<==> | <!=>) EqualityExpression

ImpliesExpression → LogicalOrExpression
                  | LogicalOrExpression ( (==> | <== ) LogicalOrExpression )+

LogicalOrExpression → LogicalAndExpression
                    | LogicalAndExpression || LogicalOrExpression

LogicalAndExpression → EqualityExpression
                     | EqualityExpression && LogicalAndExpression

EqualityExpression → RelationExpression
                   | RelationExpression (== | !=) RelationExpression

RelationExpression → AdditiveExpression
                   | AdditiveExpression ( < | <= | >= | > ) AdditiveExpression

AdditiveExpression → MultExpression
                   | MultExpression (+ | -) AdditiveExpression

MultExpression → UnaryExpression
               | UnaryExpression (* | / | %) MultExpression

UnaryExpression → CastExpression | (- | !) UnaryExpression

CastExpression → DereferenceExpression | ( Type ) UnaryExpression

DereferenceExpression → PrimaryExpression | ...

PrimaryExpression → Ident | Literal | super | true | false
                  | this | null | ( Expression )
                  | JmlPrimary

JmlPrimary → \old(Expression) | \result | ...

```

Figure A.1: Fragment of the Java grammar extended with JML specifications

Appendix B

Satisfiability Modulo Theories

As discussed in Chapter 3, *satisfiability* is the problem of determining if a formula expressing a constraint has a model.

Problems as software verification, planning, scheduling, graph manipulation, and many others, can be described in terms of satisfiability. While many of these problems can be encoded as Boolean formulas and solved using Boolean SATisfiability solvers (usually called **SAT** solvers), other problems need the expressiveness of equality, arithmetic, datatype operations, arrays, and quantifiers. Such kind of problems can be handled by solvers for theory satisfiability or Satisfiability Modulo Theories, or **SMT** for short.

For instance, in formal methods involving integers, we are only interested in showing that the formula

$$\forall x \forall y (x < y \rightarrow x < y + y)$$

is true in those interpretations in which the symbol $<$ denotes the usual ordering over the integers and $+$ denotes the addition function.

Satisfiability Modulo Theories rely on the fact that a formula is valid in a theory \mathcal{T} exactly when no interpretation of \mathcal{T} satisfies the formula's negation.

Thus, SMT solvers are SAT solvers enriched with background theories (such as arithmetic, arrays, uninterpreted functions, and so on) with the goal of check the satisfiability of first-order formulas with respect to some logical theory \mathcal{T} of interest. SMT differs from the general automated deduction area in the sense that the background theory \mathcal{T} does not need to be finitely or first-order axiomatizable. Since specialized inference methods are used for each theory, these methods can be implemented in solvers that are more efficient than general purpose theorem provers.

In order to provide standard descriptions of background theories used in the SMT systems, promote common input and output languages for SMT

```

1 (forall (x Int) (forall (y Int) (implies (< x y) (< x (+ y y))))
  )

```

Listing B.1: Example of a SMT formula

solvers, and establish and make available to the research community a large library of benchmarks for SMT solvers, *Ranise and Tinelli* developed the Satisfiability Modulo Theories Library (SMT-LIB) [BST10a]. A subset of the SMTLib standard language is presented in Figure B.1 — the full language description can be found in [BST10b]. The SMT-LIB language syntax is similar to that of the LISP programming language, and every expression is an *S-expression* (either a non-parenthesis token or a — possibly empty — sequence of S-expressions enclosed in parentheses).

The codification of the formula exemplified above using the SMT-LIB language would be as shown in Listing B.1.

This standard language is now supported by dozens of solvers: *Alt-Ergo* [CCK06], *CVC3* [BT07], *MathSAT*, *Yices* [DdM06], *Z3* [dMB08a], among many others¹.

Although the term “Satisfiability modulo theories” born less than 10 years ago, its impact on the industry is significant and can be measured by the use of SMT solvers: Microsoft uses *Z3*; Intel is using solvers such as *MathSAT* and *Boolector* for processor verification and hardware equivalence checking; other companies that are known to use SMT solvers include Galois Connection, Praxis, GrammaTech, NVIDIA, Synopsis, MathWords, etc. SMT solvers are now engines in numerous industrial applications, some of which (like scheduling) are outside the scope of deductive reasoning and formal verification — the original home community.

¹For a more complete list of SMT solvers please check at <http://smtlib.org/>, Solvers entry.

```

benchmark → ( benchmark name attribute+ )

attribute → :logic logic_name
           | :assumption formula
           | :formula formula
           | :extrafuns ( fun_symb_decl+ )
           | :extrapreds ( pred_symb_decl+ )

formula  → ( connective formula+ )
           | ( quant_symb quant_var+ formula )
           | ( let ( var term ) formula )
           | ( flet ( fvar formula ) formula )

connective → not | implies | if_then_else
            | and | or | xor | iff

quant_symb → exists | forall

quant_var  → ( var sort_symb )

```

Figure B.1: Fragment of the SMT-LIB grammar

Bibliography

- [ABB⁺05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
- [AdCHP11] Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. GammaPolarSlicer. *Computer Science and Information Systems*, 2011.
- [ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Symposium on Testing, Analysis, and Verification*, pages 60–73, 1991.
- [ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.
- [AFPMdS11] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous Software Development*. Springer, 2011.
- [Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 302–312, New York, NY, USA, 1994. ACM.
- [AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.
- [ALS06] Ali Almosawwi, Kelvin Lim, and Tanmay Sinha. Analysis tool evaluation: Coverity prevent. <http://www.cs.cmu.edu/~aldrich/courses/654/tools/cure-coverity-06.pdf>, May 2006.

- [Ame02] Peter Amey. Correctness by construction: Better can also be cheaper. *Crosstalk Magazine*, March 2002.
- [Are10] Sérgio Areias. Contracts and slicing for safety reuse. Master's thesis, Universidade do Minho, <http://www3.di.uminho.pt/~gepl/gamapolar/downloads/tese.pdf>, October 2010.
- [AT01] Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.
- [AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.
- [BA08] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. 2008.
- [Bal93] Thomas J. Ball. The use of control-flow and control dependence in software tools. Technical Report CS-TR-1993-1169, 1993.
- [Bal01] Francoise Balmas. Displaying dependence graphs: a hierarchical approach. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 261–, Washington, DC, USA, 2001. IEEE Computer Society.
- [Bal02] Françoise Balmas. Using dependence graphs as a support to document programs. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 145, Washington, DC, USA, 2002. IEEE Computer Society.
- [Ban88] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.
- [Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, 1978.
- [Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, first edition, March 2003.
- [BBC⁺06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sri-ram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the*

- 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, New York, NY, USA, 2006. ACM.
- [BBF⁺01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.
- [BBI98] Price Blaine, A. Baecker, and Small Ian. *An Introduction to Software Visualization*, chapter I, pages 3—27. The MIT Press, 1998.
- [BBL05] N. Baloian, H. Breuer, and W. Luther. Algorithm visualization using concept keyboards. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 7–16, New York, NY, USA, 2005. ACM.
- [BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.
- [BCC⁺02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. pages 85–108, 2002.
- [BCC⁺03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM.
- [BCC⁺05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

- [BCD⁺05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
- [BCF⁺10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010.
- [BCH⁺01] Adam Buchsbaum, Yih-Farn Chen, Huale Huang, Eleftherios Koutsosios, John Mocenigo, Anne Rogers, Michael Jankowsky, and Spiros Mancoridis. Visualizing and analyzing software infrastructures. *IEEE Softw.*, 18:62–70, September 2001.
- [BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004)*, pages 1–20. Springer, 2004.
- [BCM⁺92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10²⁰ states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
- [BdCP⁺08] Mario Béron, Daniela da Cruz, Maria João Varanda Pereira, Pedro Rangel Henriques, and Roberto Uzal. Evaluation criteria of software visualization systems used for program comprehension. In Universidade de Évora, editor, *Interacção'08 – 3ª Conferência Interacção Pessoa-Máquina*, October 2008.
- [BDF⁺08] Mike Barnett, Robert Deline, Manuel Fähndrich, Bart Jacobs, K. Rustan Leino, Wolfram Schulte, and Herman Venter. Verified software: Theories, tools, experiments. chapter The Spec# Programming System: Challenges and Directions, pages 144–152. Springer-Verlag, Berlin, Heidelberg, 2008.
- [BDG⁺04a] Dave Binkley, Sebastian Danicic, Tibor Gyimothy, Mark Harman, Akos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. *scam*, 00:43–52, 2004.

- [BDG⁺04b] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Form. Methods Syst. Des.*, 25(2-3):167–198, 2004.
- [BDG⁺06] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1):23–41, 2006.
- [BDW05] Michael Burch, Stephan Diehl, and Peter Weissgerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis ’05, pages 37–46, New York, NY, USA, 2005. ACM.
- [BE93] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *ICSE ’93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [BE94] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *VL*, pages 288–295, 1994.
- [BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29:33–43, April 1996.
- [Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
- [Bei95] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.
- [Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE ’07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.
- [BFH⁺08] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.
- [BFS⁺02] A. Beszedes, C. Farago, Z. Szabo, J. Csirik, and T. Gyimothy. Union slices for program maintenance, 2002.

- [BG96] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.
- [BG02] Árpád Beszédes and Tibor Gyimóthy. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–20, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.
- [BGH07] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8, 2007.
- [BGL93] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. *SIGPLAN Not.*, 28(10):65–82, 1993.
- [BH93a] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging*, pages 206–222, 1993.
- [BH93b] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.
- [BH03] David Binkley and Mark Harman. Results from a large scale study of performance optimization techniques for source code analyses based on graph reachability algorithms, 2003.
- [BHJM07] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.
- [BHPV00] Guillaume Brat, Klaus Havelund, Seungjoon Park, and Willem Visser. Java pathfinder - second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.
- [BHR95] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.
- [BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

- [Bin93] David Binkley. Precise executable interprocedural slices. *LO-PLAS*, 2(1-4):31–45, 1993.
- [Bin97] David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.
- [Bin98] David Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594, 1998.
- [Bin07] David Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.
- [BJE88] D. Bjorner, Neil D. Jones, and A. P. Ershov, editors. *Partial Evaluation and Mixed Computation*. Elsevier Science Inc., New York, NY, USA, 1988.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [BM85] Robert S. Boyer and J Strother Moore. Program verification. *Journal of Automated Reasoning*, 1:17–23, 1985.
- [BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, 2001.
- [BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems, 2004.
- [BO93] James M. Bieman and Linda M. Ott. Measuring functional cohesion. Technical Report CS-93-109, Fort Collins, CO, USA, 24 June 1993.
- [BP02] Don Box and Ted Pattison. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, London, UK, 2001. Springer-Verlag.

- [BR00a] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. pages 113–130. Springer, 2000.
- [BR00b] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Corporation, 2000.
- [BR00c] Thomas Ball and Sriram K. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in Verification*, 2000.
- [BR00d] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.
- [BRLS04] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, volume 3362, pages 49–69. Springer, Berlin, March 2004.
- [Bro77] Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737 – 751, 1977.
- [Bro88] M. H. Brown. Perspectives on algorithm animation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '88, pages 33–38, New York, NY, USA, 1988. ACM.
- [Bro92] Marc Brown. Zeus: A system for algorithm animation and multi-view editing. In *In IEEE Workshop on Visual Languages*, pages 4–9, 1992.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, 1984.
- [BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

- [BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
- [BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, Julho 2007. Berlin, Germany.
- [CA04] Gregory Conti and Kulsoom Abdullah. Passive visual fingerprinting of network attack tools. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, VizSEC/DMSEC '04, pages 45–54, New York, NY, USA, 2004. ACM.
- [CAB⁺98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, 24(7):498–520, 1998.
- [Cau10] Andrew H. Caudwell. Gource: visualizing software version control history. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 73–74, New York, NY, USA, 2010. ACM.
- [CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM.
- [CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.
- [CC93] T. Y. Chen and Y. Y Cheung. Dynamic program dicing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 378–385, Washington, DC, USA, 1993. IEEE Computer Society.

- [CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.
- [CCK06] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.
- [CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.
- [CCLL94] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 424–433, Washington, DC, USA, 1994. IEEE Computer Society.
- [CCM09] Geraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for c programs. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:123–124, 2009.
- [CCO01] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Flaver: A finite state verification technique for software systems title2:. Technical report, Amherst, MA, USA, 2001.
- [CDFP00] Pierluigi Crescenzi, Camil Demetrescu, Irene Finocchi, and Rossella Petreschi. Reversible execution and visualization of programs with leonardo. *LEONARDO, Journal of Visual Languages and Computing*, 2000.
- [CDH⁺09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09*, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [CF94] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, 1994.

- [CFR⁺99] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.
- [CGH⁺95] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. *Form. Methods Syst. Des.*, 6(2):217–232, 1995.
- [CGM⁺98] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mongardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with spin: An application to a railway interlocking system. In *SAFECOMP '98: Proceedings of the 17th International Conference on Computer Safety, Reliability and Security*, pages 284–295, London, UK, 1998. Springer-Verlag.
- [CH96] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.
- [CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *in SPIN*, pages 75–90. Springer, 2005.
- [CKSY04] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ansi-c programs using sat. *Form. Methods Syst. Des.*, 25(2-3):105–127, 2004.
- [CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 368–371, New York, NY, USA, 2003. ACM.
- [CLLF94] Gerardo Canfora, Andrea De Luccia, Giuseppe Di Lucca, and A. R. Fasolino. Slicing large programs to isolate reusable functions. In *Proceedings of EUROMICRO Conference*, pages 140–147. IEEE CS Press, 1994.
- [CLM95] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Identifying reusable functions using specification driven program

- slicing: a case study. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 124, Washington, DC, USA, 1995. IEEE Computer Society.
- [CLM96] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8(3):145–178, 1996.
- [CLYK01] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.
- [CP07] Gerardo CanforaHarman and Massimiliano Di Penta. New frontiers of reverse engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.
- [CS00] Chen and Skiena. A case study in genome-level fragment assembly. *BIOINF: Bioinformatics*, 16, 2000.
- [D'A10] Marco D'Ambros. Commit 2.0: enriching commit comments with visualization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 529–530, New York, NY, USA, 2010. ACM.
- [Dar86] Ian F. Darwin. *Checking C programs with lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1986.
- [dCHP07] Daniela da Cruz, Pedro Rangel Henriques, and Maria João Varanda Pereira. Constructing program animations using a pattern-based approach. *Comput. Sci. Inf. Syst.*, 4(2):99–116, 2007.
- [DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.
- [DdM06] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at <http://yices.csl.sri.com/tool-paper.pdf>, August 2006.

- [DG02] Stephan Diehl and Carsten Görg. Graphs, they are changing. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 23–30, London, UK, 2002. Springer-Verlag.
- [DGK00] Stephan Diehl, Carsten Goerg, and Andreas Kerren. Foresighted graphlayout. Technical report, Universität des Saarlandes, 2000.
- [DGS92] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Rigorous data flow testing through output influences. In *Proceeding on Second Irvine Software Symposium*, pages 131–145, 1992.
- [DHR⁺07] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.
- [Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.
- [Dij76a] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
- [Dij76b] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1976.
- [DKN01] Yunbo Deng, Suraj C. Kothari, and Yogy Namara. Program slice browser. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, page 50, Washington, DC, USA, 2001. IEEE Computer Society.
- [DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.
- [DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Softw. Eng.*, 31:75–90, January 2005.

- [DLB04] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–, Washington, DC, USA, 2004. IEEE Computer Society.
- [DLH04] Sebastian Danicic, Andrea De Lucia, and Mark Harman. Building executable union slices using conditioned slicing. *iwpc*, 00:89, 2004.
- [dMB08a] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.
- [dMB08b] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, 2008.
- [dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement*, chapter An Introduction to Hoare Logic, pages 363–386. Cambridge University Press, 1998.
- [EGK⁺02] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Trans. Softw. Eng.*, 28:396–412, April 2002.
- [EKS] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 602–611, Washington, DC, USA. IEEE Computer Society.
- [EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.
- [ESS99] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Readings in information visualization. chapter Seesofta tool for visualizing line oriented software statistics, pages 419–430. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
- [FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 2103–2110, New York, NY, USA, 2010. ACM.

- [FdCHV08] Rúben Fonseca, Daniela da Cruz, Pedro Henriques, and Maria João Varanda. How to interconnect operational and behavioral views of web applications. In *ICPC '08: IEEE International Conference on Program Comprehension*, Amsterdam, The Netherlands, 2008. IEEE Computer Society.
- [FDHH01] Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 89–97. IEEE Computer Society, 2001.
- [FDJ98] Loe Feijs and Roel De Jong. 3D visualization of software architectures. *Commun. ACM*, 41:73–78, December 1998.
- [FG97] Istvan Forgács and Tibor Gyimóthy. An Efficient Interprocedural Slicing Method for Large Programs. Technical Report TR97-7, Informatics Lab., MTA SZTAKI, June 1997.
- [FG02] Csaba Faragó and Tamás Gergely. Handling pointers and unstructured statements in the forward computed dynamic slice algorithm. *Acta Cybern.*, 15(4):489–508, 2002.
- [FGKS91] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 317–326, New York, NY, USA, 1991. ACM.
- [FHK⁺02] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kerny Wong. The software bookshelf. pages 295–339, 2002.
- [FLL⁺02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37:234–245, May 2002.
- [Flo67] Robert Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.
- [FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

- [FN87] William B. Frakes and Brian A. Nejme. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
- [FP11] Maria João Frade and Jorge Sousa Pinto. Verification Conditions for Source-level Imperative Programs. *Computer Science Review*, 5:252–277, 2011.
- [FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a compiler*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.
- [FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, New York, NY, USA, 1995. ACM.
- [FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, USA, 2001. ACM.
- [FSKG92] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.
- [GABF99] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 303–321, London, UK, 1999. Springer-Verlag.
- [Gal90] Keith Brian Gallagher. *Using program slicing in software maintenance*. PhD thesis, Catonsville, MD, USA, 1990.

- [Gal96] Keith Gallagher. Visual impact analysis. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 52–58, Washington, DC, USA, 1996. IEEE Computer Society.
- [Gal04] Daniel Galin. *Software Quality Assurance*. Addison Wesley, Harlow, England, 2004.
- [Gan09] Jack Ganssle. A new os has been proven to be correct using mathematical proofs. the cost: astronomical. <http://www.embedded.com/columns/breakpoint/220900551>, November 2009.
- [GC95] G. C. Gannod and B. H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *Proceedings of the Second Working Conference on Reverse Engineering, WCRE '95*, pages 166–, Washington, DC, USA, 1995. IEEE Computer Society.
- [GC10] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer London, 2010.
- [Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen II. *Mathematische Zeitschrift*, 39, 1935.
- [Ger91] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.
- [GFR06] João Gama, Ricardo Fernandes, and Ricardo Rocha. Decision trees for mining data streams. *Intell. Data Anal.*, 10(1):23–45, 2006.
- [GHS92] R. Gupta, M. Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 299–308, 1992.
- [GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99*, pages 99–, Washington, DC, USA, 1999. IEEE Computer Society.

- [GKSD05] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society.
- [GL91b] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.
- [GM02] D. Goswami and R. Mall. An efficient method for computing dynamic program slices. *Inf. Process. Lett.*, 81(2):111–117, 2002.
- [GME05] Denis Gracanin, Kresimir Matkovic, and Mohamed El-toweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1:221–230, 2005. 10.1007/s11334-005-0019-8.
- [GMMS07] Ulrich Güntzer, Rudolf Müller, Stefan Müller, and Ralf-Dieter Schimkat. Retrieval for decision support resources by structured models. *Decis. Support Syst.*, 43(4):1117–1132, 2007.
- [GO97] Keith Gallagher and Liam O’Brien. Reducing visualization complexity using decomposition slices. In *Proc. Software Visualisation Work.*, pages 113–118, Adelaide, Australia, 11–12 Dezembro 1997. Department of Computer Science, Flinders University.
- [God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL ’97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.
- [Gop91] R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Software Maintenance’91 Conference*, pages 191–200, Sorrento, Italy, October 1991.
- [Gra08a] GramaTech. A code-analysis tool that identifies complex bugs at compile time. <http://www.grammatech.com/products/codesonar/>, 2008.
- [Gra08b] GramaTech. A code browser that understands pointers, indirect function calls, and whole-program effects. <http://www.grammatech.com/products/codesurfer/>, 2008.

- [GSH97] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, 1997.
- [Hal95] R.J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995. An algorithm to automatically extract a correctly functioning subset of the code of a system is presented. The technique is based on computing a simultaneous dynamic program slice of the code for a set of representative inputs. Experiments show that the algorithm produces significantly smaller subsets than with existing methods.
- [Ham04] Paul Hamill. *Unit test frameworks*. O'Reilly, 2004.
- [Har00] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.
- [HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *J. Syst. Softw.*, 68(1):45–64, 2003.
- [HBP02] Dixie Hisley, Matthew J. Bridges, and Lori L. Pollock. Static interprocedural slicing of shared memory parallel programs. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 658–664. CSREA Press, 2002.
- [HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification & Reliability*, 5(3):143–162, 1995.
- [HD98] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.
- [HEH⁺98] Jean Henrard, Vincent Englebert, Jean-Marc Hick, Didier Roland, and Jean-Luc Hainaut. Program understanding in databases reverse engineering. In *Database and Expert Systems Applications*, pages 70–79, 1998.
- [HHF⁺01] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. pages 138–147, 2001.

- [HHF⁺02] Robert M. Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Softw. Test., Verif. Reliab.*, pages 23–28, 2002.
- [HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Höglström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.
- [HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast, 2003.
- [HK] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*.
- [HKM07] Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: development tools for security-typed languages. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 1–10, New York, NY, USA, 2007. ACM.
- [HLL⁺09] John Hatchliff, Gary Leavens, Rustan Leino, Peter Muller, and Mathew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, 2009.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [Hoe05] Urs Hoelzle. Google: or how i learned to love terabytes. *SIG-METRICS Perform. Eval. Rev.*, 33(1):1–1, 2005.
- [HOSD] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of function coupling.
- [HP98] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder, 1998.
- [HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, San Diego, California, 1988.

- [HPR89a] Susan Horwitz, P. Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 28–40, New York, NY, USA, 1989. ACM.
- [HPR89b] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.
- [HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM.
- [HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.
- [HS97] D. Huynh and Y. Song. Forward computation of dynamic slicing in the presence of structured jump statements. In *Proceedings of ISACC*, pages 73–81, 1997.
- [HSD98] Mark Harman, Yoga Sivagurunathan, and Sebastian Danicic. Analysis of dynamic memory access using amorphous slicing. In *ICSM*, pages 336–, 1998.
- [HSS01] Uri Hanani, Bracha Shapira, and Peretz Shoval. Information filtering: Overview of issues, research and systems. *User Modeling and User-Adapted Interaction*, 11(3):203–259, 2001.
- [Hun02] C. Hundhausen. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, Junho 2002.
- [ICG07] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.
- [Inf09] AbsInt Angewandte Informatik. aiT WCET Analyzers. <http://www.absint.com/ait/>, December 2009.
- [Ins09] Klocwork Insight. Klocwork insight - proven source code analysis. www.klocwork.com, December 2009.

- [Jaz07] Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: 2007 Future of Software Engineering*, pages 199–213, Washington, DC, USA, 2007. IEEE Computer Society.
- [JB10] Ivan Jager and David Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, 2010.
- [JDC88] Hwang J.C., M.W. Du, and C.R. Chou. Finding program slices for recursive procedures. In *Proceedings of IEEE COMPSAC 88*, Washington, DC, 1988. IEEE Computer Society.
- [JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):1–54, 2009.
- [Joh78] Stephen Johnson. Lint, a C program checker, 1978.
- [JR94] Daniel Jackson and Eugene J. Rollins. A new model of program dependence for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM.
- [JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM.
- [JS98] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4:257–271, July 1998.
- [Jun07] Dieter Jungnickel. *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd edition, 2007.
- [JZR91] J. Jiang, X. Zhou, and D.J. Robson. Program slicing for C - the problems in implementation. In *Proceedings of Conference on Software Maintenance*, pages 182–190. IEEE CPress, 1991.
- [KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell,

- Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.
- [Kei02] Daniel A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8:1–8, January 2002.
- [KFN99] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [KFS93a] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceeding on Conference on Software Maintenance*, pages 386–395, 1993.
- [KFS93b] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocess. Microprogram.*, 38(1-5):625–636, 1993.
- [KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–56, 2001.
- [KH02] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 96–112, London, UK, 2002. Springer-Verlag.
- [Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [KJLG03] Akos Kiss, Judit Jasz, Gabor Lehotai, and Tibor Gyimothy. Interprocedural static slicing of binary executables. *scam*, 00:118, 2003.
- [KKM06] Ville Karavirta, Ari Korhonen, and Lauri Malmi. Taxonomy of algorithm animation languages. In *Proceedings of the 2006 ACM symposium on Software visualization, SoftVis '06*, pages 77–85, New York, NY, USA, 2006. ACM.
- [KKMN10] Ville Karavirta, Ari Korhonen, Lauri Malmi, and Thomas Naps. A comprehensive taxonomy of algorithm animation languages. *J. Vis. Lang. Comput.*, 21:1–22, February 2010.

- [KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
- [KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.
- [Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
- [KLM91] J. Schwalbe K. L. McMillan. Formal verification of the encore gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, 1991.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242. SpringerVerlag, 1997.
- [KM99] Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. In *Proceedings of the 7th International Workshop on Program Comprehension, IWPC '99*, pages 4–, Washington, DC, USA, 1999. IEEE Computer Society.
- [Kor97a] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Trans. Softw. Eng.*, 23(1):17–34, 1997.
- [Kor97b] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, January 1997.
- [Kos02] Rainer Koschke. Software visualization for reverse engineering. In *Revised Lectures on Software Visualization, International Seminar*, pages 138–150, London, UK, 2002. Springer-Verlag.
- [KR97] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. *wpc*, 0:80, 1997.
- [Kri04] Jens Krinke. Visualization of program dependence and slices. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.
- [KRML04] Nick Kidd, Thomas Reps, David Melski, and Akash Lal. Wpds++: A c++ library for weighted pushdown systems, 2004.

- [Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.
- [KS06] Sarfraz Khurshid and Yuk Lai Suen. Generalizing symbolic execution to library classes. *SIGSOFT Softw. Eng. Notes*, 31(1):103–110, 2006.
- [KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384, London, UK, 1992. Springer-Verlag.
- [KT04] Yiannis Kanellopoulos and Christos Tjortjis. Data mining source code to facilitate program comprehension: Experiments on clustering data retrieved from c++ programs. *iwpc*, 00:214, 2004.
- [KY94] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 66–79, New York, NY, USA, 1994. ACM.
- [Lak92] Arun Lakhotia. Improved interprocedural slicing algorithm, 1992.
- [Lak93] Arun Lakhotia. Rule-based approach to computing module cohesion. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 35–44, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977.
- [Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [LBO⁺07] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, New York, NY, USA, 2007. ACM.

- [LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.
- [LC94] Panos E. Livadas and Stephen Croll. System dependence graphs based on parse trees and their use in software maintenance. *Information Sciences*, 76(3-4):197–232, 1994.
- [LC03] G. Leavens and Y. Cheon. Design by Contract with JML, 2003.
- [LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering, ICSE '05*, pages 672–673, New York, NY, USA, 2005. ACM.
- [LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.
- [Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
- [LFB06] Dawn J. Lawrie, Henry Feild, and David Binkley. Leveraged quality assessment using information retrieval techniques. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 149–158, Washington, DC, USA, 2006. IEEE Computer Society.
- [LFM96] Andrea De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 9–18, 1996.
- [LFY⁺06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32:831–848, 2006.
- [LHHK03] Andrea De Lucia, Mark Harman, Robert Hierons, and Jens Krinke. Unions of slices are not slices. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2003)*, 2003.

- [LJ80] George Lakoff and Mark Johnson. *Metaphors We Live By*. University Of Chicago Press, first edition edition, April 1980.
- [LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 285, Washington, DC, USA, 2003. IEEE Computer Society.
- [LMFB06] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
- [LMS08] K. Rustan M. Leino, Michal Moskal, and Wolfram Schulte. Verification condition splitting. Microsfot Research, 2008.
- [LMS09] K. Rustan Leino, Peter Müller, and Jan Smans. *Verification of Concurrent Programs with Chalice*, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009.
- [Lon85] H. D. Longworth. Slice-based program metrics. Master's thesis, 1985.
- [Lop08] Crista Lopes. Codegenie. <http://sourcerer.ics.uci.edu/codegenie/>, 2008.
- [LR87] Hareton K. N. Leung and Hassan K. Reghbat. Comments on program slicing. *IEEE Trans. Softw. Eng.*, 13(12):1370–1371, 1987.
- [LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.
- [LSL96] Hongjun Lu, Rudy Setiono, and Huan Liu. Effective data mining using neural networks. *IEEE Trans. on Knowl. and Data Eng.*, 8(6):957–961, 1996.
- [LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.
- [Ltd92] TA Consultancy Services Ltd. Malpas training course. TACS/9093/15, August 1992.

- [Luc01] Andrea De Lucia. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.
- [LV97] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, April 1997.
- [LW86] Jim Lyle and Mark Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the 1st Conference on Empirical Studies of Programming*, pages 187–197, Norwood, New Jersey, 1986. Ablex publishing.
- [LW87] Jim Lyle and Mark Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, 1987.
- [LWG⁺95] Jim Lyle, D. Wallace, J. Graham, Keith Gallagher, J. Poole, and David Binkley. Unravel: A case tool to assist evaluation of high integrity software, 1995.
- [Lyl84] James Robert Lyle. *Evaluating variations on program slicing for debugging (data-flow, ada)*. PhD thesis, College Park, MD, USA, 1984.
- [Ma04] Kwan-Liu Ma. Visualization for security. *SIGGRAPH Comput. Graph.*, 38:4–6, November 2004.
- [Mac10] Matthew MacDonald. *Pro Silverlight 4 in C#*. Pro to Expert Series. Apress, 2010.
- [Mar93] Kamkar Mariam. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Linköping University, Sweden, 1993.
- [Mar03] Andrian Marcus. *Semantic-driven program analysis*. PhD thesis, Kent, OH, USA, 2003. Director-Jonathan I. Maletic.
- [MC88] B. P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Not.*, 23(7):135–144, 1988.
- [McG82] Andrew D. McGettrick. *Program Verification Using ADA*. Cambridge University Press, New York, NY, USA, 1982.
- [McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

- [ME04] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *In Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 155–168, 2004.
- [Mer00] Stephan Merz. Model checking: A tutorial overview. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2000.
- [Mey86] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.
- [Mey87] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22:85–94, February 1987.
- [Mey92] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, 1992.
- [MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. Comprehension of software analysis data using 3d visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension, IWPC ’03*, pages 105–, Washington, DC, USA, 2003. IEEE Computer Society.
- [Mic08a] Microsoft. Design guidelines for class library developers. [http://msdn.microsoft.com/en-us/library/czefa0ke\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/czefa0ke(VS.71).aspx), 2008.
- [Mic08b] Microsoft. Fxcop. [http://msdn.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx), 2008.
- [MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.
- [MLMD01] Jonathan I. Maletic, Jason Leigh, Andrian Marcus, and Greg Dunlap. Visualizing object-oriented software in virtual reality. In *Proceedings of the 9th international workshop on program comprehension (IWPC’01)*, pages 26–35, Washington, DC, USA, 2001. IEEE Computer Society.
- [MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE ’01: Proceedings of the 16th IEEE international conference on Automated*

- software engineering*, page 107, Washington, DC, USA, 2001. IEEE Computer Society.
- [MM11] Yannick Moy and Claude Marché. Jessie plugin tutorial. <http://frama-c.com/download/jessie-tutorial-Carbon-20101202-beta2.pdf>, February 2011. The official manual for the Frama-C plug-in Jessie.
- [MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. Atask oriented view of software visualization. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–, Washington, DC, USA, 2002. IEEE Computer Society.
- [MMCG99] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 50, Washington, DC, USA, 1999. IEEE Computer Society.
- [MMS03] G. B. Mund, Rajib Mall, and S. Sarkar. Computation of intraprocedural dynamic program slices. *Information & Software Technology*, 45(8):499–512, 2003.
- [MPC⁺02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.
- [MSABA10] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM.
- [MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.
- [MTO⁺92] Hausi A. Müller, Scott R. Tilley, Mehmet A. Orgun, B. D. Corrie, and Nazim H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection

- models. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992.
- [MX05] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.
- [Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1:97–123, March 1990.
- [MZZ⁺01] Andrew Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. <http://www.cs.cornell.edu/jif>, July 2001. Date of information: December 2009.
- [NAS09] NASA. C global surveyor. <http://ti.arc.nasa.gov/project/cgs/>, December 2009.
- [NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8:12–26, August 1973.
- [OB92] Linda Ottenstein and James Bieman. Effects of software changes on module cohesion. In *International Conference on Software Maintenance*, pages 345–353, 1992.
- [MDUK96] Ministry of Defense of United Kingdom. Safety management requirements for defense systems. <http://www.savive.com/external/ukmod/defstan/defstan0056/index.html>, January 1996.
- [IEEE90] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Press, 1990.
- [OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, 1984.
- [OPdCB10] Nuno Oliveira, Maria Joao Varanda Pereira, Daniela da Cruz, and Mario Beron. Influence of synchronized domain visualizations on program comprehension. *International Conference on Program Comprehension*, 0:192–195, 2010.
- [OSH01] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *ICSM*, pages 158–, 2001.

- [OT89] Linda Ottenstein and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *International Conference on Software Engineering*, pages 198–204, 1989.
- [OT93] Linda Ottenstein and J. Thuss. Slice based metrics for estimating cohesion, 1993.
- [Ott92] Linda Ottenstein. Using slice profiles and metrics during software maintenance, 1992.
- [Pac04] Michael J. Pacione. Software visualisation for object-oriented program comprehension. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 63–65, Washington, DC, USA, 2004. IEEE Computer Society.
- [Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
- [PBS93] B. Price, R. Baecker, and I. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
- [PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In *Proceedings of the Symposium on Graph Drawing*, GD '95, pages 435–446, London, UK, 1996. Springer-Verlag.
- [PD95] Si Pan and R. Geoff Dromey. Using Strongest Postconditions to Improve Software Quality. In *Software Quality and Productivity: Theory, practice and training*, pages 235–240, London, UK, UK, 1995. Chapman & Hall, Ltd.
- [PJNR⁺98] Lacan Philippe, Monfort Jean N., Le Vinh Q. Ribal, Alain Deutsch, and Georges Gonthier. Ariane 5 - the software reliability verification process. *DASIA 98 - Data Systems in Aerospace, Proceedings of the conference held 25-28 May, 1998 in Athens, Greece*, 1998.
- [PM04] Sankar K. Pal and Pabitra Mitra. *Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing*. Chapman & Hall, Ltd., London, UK, UK, 2004.
- [PR98] Willard C. Pierson and Susan H. Rodger. Web-based animation of data structures using jawaa. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on*

- Computer science education*, pages 267–271, New York, NY, USA, 1998. ACM.
- [Pre86] Roger S. Pressman. *Software engineering: a practitioner's approach (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1986.
- [PV06] Sokhom Pheng and Clark Verbrugge. Dynamic data structure analysis for java programs. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.
- [PV09] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11:339–353, October 2009.
- [QH04a] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in jvms. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 11–11, Berkeley, CA, USA, 2004. USENIX Association.
- [QH04b] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in jvms. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 11–11, Berkeley, CA, USA, 2004. USENIX Association.
- [QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [RB06] Nuno Rodrigues and Luis Soares Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.
- [RBF96] Ronald B. Finkbine. Metrics and models in software quality engineering. *SIGSOFT Softw. Eng. Notes*, 21(1):89, 1996.
- [RBL06] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In

- PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.
- [RC92] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: the art of mapping programs to pictures. In *Proceedings of the 14th international conference on Software engineering, ICSE '92*, pages 412–420, New York, NY, USA, 1992. ACM.
- [RC93] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *Computer*, 26:11–24, December 1993.
- [RCB⁺03] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher, II, and M. Main. End-user software visualizations for fault localization. In *Proceedings of the 2003 ACM symposium on Software visualization, SoftVis '03*, pages 123–132, New York, NY, USA, 2003. ACM.
- [RD06] Daniel Ratiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 83–92, Washington, DC, USA, 2006. IEEE Computer Society.
- [Rep96] Thomas W. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, 1996.
- [RHJ⁺06] Ansgar Fehnker Ralf, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna — a static model checker, 2006.
- [RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.
- [RM05] Juergen Rilling and S. P. Mudur. 3d visualization techniques to support slicing-based program comprehension. *Comput. Graph.*, 29:311–329, June 2005.
- [Rod06] Nuno Rodrigues. Haslicer. <http://labdotnet.di.uminho.pt/Haslicer/Haslicer.aspx>, 2006.

- [RSF00] Guido Rössling, Markus Schüer, and Bernd Freisleben. The animal algorithm animation tool. *SIGCSE Bull.*, 32(3):37–40, 2000.
- [RSJM05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Mel-ski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
- [Rug95] S. Rugaber. Program comprehension, 1995.
- [San95] Georg Sander. Graph layout through the vcg tool. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 194–205, London, UK, 1995. Springer-Verlag.
- [Sar03] Kamran Sartipi. Software architecture recovery based on pattern matching. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 293, Washington, DC, USA, 2003. IEEE Computer Society.
- [SB08] Amit Prakash Sawant and Naveen Bali. Multivizarch: multiple graphical layouts for visualizing software architecture. *Crossroads*, 14:17–21, March 2008.
- [Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM.
- [Sch02] David A. Schmidt. Structure-preserving binary relations for program abstraction. pages 245–265, 2002.
- [SDBP98] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.
- [SFFG10] Lajos Schrettner, Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Visualization of software architecture graphs of java systems: managing propagated low level dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 148–157, New York, NY, USA, 2010. ACM.
- [SHR99] Saurabh Sinha, Mary Jean Harrold, and Gregg Roethermel. System-dependence-graph-based slicing of programs with ar-

- bitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.
- [SHS02] Yoga Sivagurunathan, Mark Harman, and Bala Sivagurunathan. Slice-based dynamic memory modelling – a case study, 2002.
- [Sil06] Josep Silva. A comparative study of algorithmic debugging strategies. In *Proceedings of the 16th international conference on Logic-based program synthesis and transformation*, LOPSTR’06, pages 143–159, Berlin, Heidelberg, 2006. Springer-Verlag.
- [SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-language program analysis and refactoring. In *SCAM ’06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.
- [SM95] M.-A. D. Storey and H. A. Muller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’95, pages 275–, Washington, DC, USA, 1995. IEEE Computer Society.
- [SMS01] Timothy S. Souder, Spiros Mancoridis, and Maher Salah. Form: A framework for creating views of program executions. In *ICSM*, pages 612–, 2001.
- [SR03] Eric J. Stierna and Neil C. Rowe. Applying information-retrieval methods to software reuse: a case study. *Inf. Process. Manage.*, 39(1):67–74, 2003.
- [SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
- [Sta90] John T. Stasko. Simplifying algorithm animation with tango. In *VL*, pages 1–6, 1990.
- [Ste91] Bernhard Steffen. Data flow analysis as model checking. In *TACS ’91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 346–365, London, UK, 1991. Springer-Verlag.
- [Sto96] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

- [Thu88] J.J. Thuss. An investigation into slice based cohesion metrics. Master's thesis, 1988.
- [Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
- [TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.
- [Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM.
- [VRD04] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 328–337, Washington, DC, USA, 2004. IEEE Computer Society.
- [Wal91] David W. Wall. Systems for late code modification. In *Code Generation*, pages 275–293, 1991.
- [War09] Martin Ward. Properties of slicing definitions. In *SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 23–32, Washington, DC, USA, 2009. IEEE Computer Society.
- [WC96] Norman Wilde and Christopher Casey. Early field experience with the software reconnaissance technique for program comprehension. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 312–318, Washington, DC, USA, 1996. IEEE Computer Society.
- [WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–310, New York, NY, USA, 1994. ACM.
- [Wei79] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979.

- [Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
- [WFP07] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.
- [WL86] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
- [WL07] Richard Wettel and Michele Lanza. Visualizing software systems as cities. *Visualizing Software for Understanding and Analysis, International Workshop on*, 0:92–99, 2007.
- [WR08] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30(2):1–49, 2008.
- [WRG] Tao Wang, Abhik Roychoudhury, and Liang Guo. Jslice. <http://jslice.sourceforge.net/>.
- [XA05] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, 2005.
- [XPM06] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. Visualization of cvs repository information. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.
- [YM98] Peter Young and M. Munro. Visualizing software in virtual reality. In *Proceedings of the 6th International Workshop on Program Comprehension, IWPC '98*, pages 19–, Washington, DC, USA, 1998. IEEE Computer Society.
- [Yoo04] InSeon Yoo. Visualizing windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security, VizSEC/DMSEC '04*, pages 82–89, New York, NY, USA, 2004. ACM.

- [You99] Peter Young. *Visualising Software in Cyberspace*. PhD thesis, University of Durham, 1999.
- [YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.
- [YVQ⁺10] Xiaohong Yuan, Percy Vega, Yaseen Qadah, Ricky Archer, Huiming Yu, and Jinsheng Xu. Visualization tools for teaching computer security. *Trans. Comput. Educ.*, 9:20:1–20:28, January 2010.
- [ZB04] Jihong Zeng and Peter A. Bloniarz. From keywords to links: an automatic approach. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 283, Washington, DC, USA, 2004. IEEE Computer Society.
- [Zel01] Andreas Zeller. Automated debugging: Are we close. *Computer*, 34(11):26–31, 2001.
- [Zel07] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *FOSE '07: 2007 Future of Software Engineering*, pages 316–325, Washington, DC, USA, 2007. IEEE Computer Society.
- [ZG04] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing, 2004.
- [ZGZ04] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Washington, DC, USA, 2004. IEEE Computer Society.
- [Zha03] Kang Zhang, editor. *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.