Infeasible Path Detection : a Formal Model and an Algorithm

Romain Aïssat

► To cite this version:

Romain Aïssat. Infeasible Path Detection : a Formal Model and an Algorithm. Other [cs.OH]. Université Paris Saclay (COmUE), 2017. English. NNT : 2017SACLS036 . tel-01567093

HAL Id: tel-01567093
https://theses.hal.science/tel-01567093

Submitted on 21 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2017SACLS036

Thèse de doctorat de l'Université Paris-Saclay préparée à l'Université Paris-Sud

Ecole doctorale n°580
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat : Informatique

par

M. Romain Aïssat

Infeasible Paths Detection: a Formal Model and an Algorithm

Thèse présentée et soutenue à Gif-Sur-Yvette, le 30 janvier 2017.

Composition du Jury :

M.	Alain Denise	Professeur Université Paris-Sud	(Président du jury)
Mme.	Sandrine Blazy	Professeure Université de Rennes 1	(Rapporteur)
Mme.	Lydie du Bousquet	Professeure Université Grenoble Alpes	(Rapporteur)
M.	François Laroussinie	Professeur Université Paris Diderot	(Examinateur)
M.	Frédéric Voisin	Maître de Conférence Université Paris-Sud	(Examinateur)
M.	Burkhart Wolff	Professeur Université Paris-Sud	(Directeur de thèse)
Mme.	Marie-Claude Gaudel	Professeure émérite Université Paris-Sud	(Invitée)
M.	Jean-Yves Pierron	Ingénieur-chercheur CEA-LIST	(Invité)

Acknowledgments

Conducting the works related here and writing this document was a long and hard task that would not have been possible with a number of people I would like to thank.

First, I would like to thank my main adviser, Burkhart Wolff. I was always impressed by Bu's ability to think beyond the actual problem we were tackling and really enjoyed our talks – although I could only understand a small part of them during the first weeks following my arrival at LRI. I will always smile when remembering that talk in which we argued for three whole hours about our "different" concepts of symbolic execution, only to realize that we in fact agree and, a few minutes later, end that talk laughing. Bu's great knowledge and experience were invaluable to me and I am very grateful for him to have push my investigations in some areas I was not familiar with or did not consider in the first place. Burkhart played a major role in the development of my works and helped me build an elegant solution – which I dare say I am proud of – to a difficult problem. I also had a great time teaching with Burkhart: he is a passionate teacher, always listening to his students and willing to help them.

I would also like to thank my co-adviser Marie-Claude Gaudel. Marie-Claude's long experience and great knowledge were naturally invaluable to me, and her rigour and her determination will always make her a model to me as a scientist. Although I guess we could say we had some hard times in the beginning, I was very happy and proud to eventually gain Marie-Claude's trust, and even more so seeing her so involved in my works. I am very proud of Marie-Claude's encouragements and will always be grateful for her active support in the downs of those more-than-four years at LRI.

Last but not least, I would like to thank Frédéric Voisin, also my co-adviser. Although I knew Burkhart and Marie-Claude would always help me if I had a question or a problem, Frédéric has simply always been there for me, no matter the reason, the day or the hour of the day (and very frequently of the night). Whatever the problem was – what direction give to my work, is a particular algorithm correct, what is that bug that is causing my code to fail, how should I explain this to students, how should I write this piece of text, how to handle this with the administration, where is my badge, did I broke something on my bike – I not only knew I could count on

him, but also that with his help we would solve it. I often felt guilty for the amount of work and things to read I gave him, and I will never say enough how grateful I am. His confidence in my abilities, his strong and constant support of my own ideas and research directions as well as number of his human qualities make him the best adviser a Ph.D. student could think of.

It was an honor and a pleasure to work with Burkhart, Marie-Claude and Frédéric, and I sincerely hope I will have the opportunity to interact with them again in the rest of my career.

I would then like to thanks the members of the VALS team for welcoming me in their team and for the warm and friendly atmosphere. My thoughts first go to Yakoub, Bu's other student with whom I shared my office during three years and a half. Besides the office, we also shared our pains, our joys, our failures and our successes, as well as a number of tastes and passions for all sorts of things, which naturally led to great fun and talks. Although they still owe me some cigarettes, I would also like to thank some of the other Ph.D. students of the team – Hai and Stefania – for the good times and for their constant support. Special thanks also go to Frédéric Tuong, Delphine, Thibault, Sylvain, Véronique, Evelyne, Xavier and Guillaume.

I would also like to thank my old colleagues at CEA for their constant support. I am particularly grateful to Jean-Yves Pierron for trusting me a few years ago as an internship candidate on a fascinating subject, which is what I think made it possible for me to start the works related in this document.

From the University Paris Diderot, my thanks go to Ahmed Bouajjani and François Laroussinie, who actively supported me on numerous occasions, before, during, and after those four years spent at LRI. I am also grateful to Dominique Poulhalon and Juliusz Chroboczek for their help when I had the opportunity to teach with them.

If my colleagues took an important part in conducting these works, my relatives are also to be thanked. I would like to thank in no particular order my dear friends Julien, Bruno and Laura, Alex, Stéphane, Isa, Big Fanny and Clementine, Small Fanny and André, Benjamin, Alex, Robin and Aude, Michou, Dermo, Antho, Kitty and co, the Babies, the Aurels', Laura and Fab, Lucas and Erika, Ozz and Marion, JB and Victoire, the Clements, Eliel and Noémie, Gary and Alix, Amandine as well as my cousins Cédric, Loïc, Julie, Karima, Farez, Sophiane, etc, and my beloved little brother Alexandre. Their love and friendship always pushed me forward, and I would not have been able to go this far without their support and their presence in the good and bad times. Special thanks go to Florent and Chloé, who have a bigger part in all of this than they might think.

I most certainly did not name every person I would like to thank: many people helped me in a way or another to conclude these works.

Finally, I would like to thank my parents for all they have done for me during all these years, for helping me become who I am and for everything

that helped me get to this day. I will never have enough words to express how grateful I am to them, which is a good reason to end this section now. This work is dedicated to them.

Contents

1 Introduction **13**
1.1 Software Testing 13
1.1.1 Principles of Testing 13
1.1.2 Selecting Test Cases 14
1.2 Motivations 18
1.3 Contributions 20

2 Context: Random Testing and Infeasibility **23**
2.1 Random Structural Biased Testing 23
2.1.1 Isotropic Random Walks 23
2.1.2 Uniform Random Walks 25
2.2 Program Paths and Infeasibility 27
2.3 Symbolic Execution and Unbounded Loops 30

3 Introducing Red-Black Graphs and their Transformations **35**
3.1 Introduction 35
3.2 Modeling programs 36
3.3 Operational Semantics of Programs 39
3.3.1 Configurations 39
3.3.2 Symbolic Execution Steps 42
3.3.3 Symbolic Execution of Programs 45
3.4 Subsumption 47
3.4.1 Subsumption 47
3.4.2 Abstracting Configurations 50
3.5 Red-Black Graphs 52
3.6 Red-Black Graphs Transformations 57
3.6.1 Extension by Symbolic Execution 57
3.6.2 Extension by Subsumption 58
3.6.3 Extension by Abstraction 59
3.6.4 Extension by Marking 60
3.6.5 Extension by Strengthening 61
3.6.6 The Set of Red-Black Graphs 62
3.7 Building Red-Black Graphs: an Example 62

3.8 Summary 67

4 Formalization **71**

4.1 Introduction 71
4.2 Symbolic Execution 73
4.2.1 Arithmetic and Boolean Expressions 73
4.2.2 Stores 76
4.2.3 Configurations, Subsumption and Abstraction 77
4.2.4 Symbolic Execution Steps 79
4.3 Graphs, Labeled Transition Systems, Subsumption Relations 81
4.3.1 Introduction 81
4.3.2 Rooted Graphs 82
4.3.3 Labeled Transition Systems 84
4.3.4 Graphs Equipped with Subsumption Relations 85
4.3.5 Extending Graphs and Subsumption Relations 88
4.4 Red-Black Graphs and Their Properties 91
4.4.1 The Type of Red-Black Graphs 91
4.4.2 Well-Formed Red-Black Graphs 92
4.4.3 Relation Between Red Vertices 96
4.4.4 Preservation of Behaviours 98
4.4.5 Preservation of Feasible Paths 102
4.5 Summary 104

5 Algorithm **107**

5.1 Introduction 107
5.2 Data Structures and Inputs 108
5.2.1 Data Structures 108
5.2.2 Inputs and Parameters 109
5.3 Building the Red-Black Graph 110
5.3.1 Principles 110
5.3.2 Symbolic Execution Steps 112
5.3.3 Detecting Subsumptions 113
5.3.4 Refine-and-Restart Mechanism 124
5.3.5 Look-Ahead Mechanism 126
5.3.6 Building The Resulting LTS 128
5.4 The Merging Sort Example 128
5.4.1 The Merging Sort Program 128
5.4.2 Merging Sort without Path Sets Comparisons 131
5.4.3 Merging Sort with Path Sets Comparisons 141
5.5 Summary 151

6 Experiments and Discussions 153
6.1 Experimental Results 153
6.1.1 Greatest Common Divisor 156
6.1.2 Merging Sort 158
6.1.3 Substring 161
6.1.4 Bubble Sort 164
6.1.5 Bounded Loops 166
6.1.6 Modulo Example 169
6.2 Discussions and Possible Improvements 171
6.2.1 Extending Refinements 171
6.2.2 Look-Ahead Mechanism 173
6.2.3 Abstraction methods 174
6.2.4 Subsumptions Between Different Paths 175
6.3 Summary 177

7 Conclusion 179

A Isabelle/HOL Formalization 189
A.1 Introduction 189
A.2 Arithmetic Expressions 190
A.2.1 Variables and their domain 190
A.2.2 Program and symbolic states 191
A.2.3 The *aexp* type-synonym 191
A.2.4 Variables of an arithmetic expression 192
A.2.5 Fresh variables 193
A.3 Boolean Expressions 193
A.3.1 Basic definitions 193
A.3.2 Properties about the variables of an expression 194
A.4 Stores 196
A.4.1 Basic definitions 196
A.4.2 Consistency 197
A.4.3 Adaptation of an arithmetic expression to a store . . . 198
A.4.4 Adaptation of a boolean expression to a store 201
A.5 Configurations and Subsumption 204
A.5.1 Configurations 204
A.5.2 Symbolic variables of a configuration. 204
A.5.3 Freshness. 204
A.5.4 Satisfiability 204
A.5.5 States of a configuration 205
A.5.6 Subsumption 205
A.5.7 Semantics of a configuration 206
A.5.8 Entailment 206
A.5.9 Abstractions 207
A.6 Symbolic Execution 207

A.6.1 Labels 207
A.6.2 Definitions of *SE* and *SE_star* 208
A.6.3 Basic properties of *SE* 209
A.6.4 Monotonicity of *SE* 213
A.6.5 Basic properties of *SE_star* 214
A.6.6 Monotonicity of *SE_star* 215
A.6.7 Existence of successors 216
A.6.8 Feasibility of a sequence of labels 220
A.6.9 Concrete execution 222
A.6.10 Weakest Precondition Calculus 224
A.7 Rooted Graphs 225
A.7.1 Basic definitions and properties 225
A.7.2 Consistent edge sequences, sub-paths and paths 227
A.7.3 Adding edges 230
A.8 Labeled Transition Systems 230
A.8.1 Basic definitions 230
A.8.2 Feasible sub-paths and paths 232
A.9 Graphs Equipped with Ssubsumption Relations 233
A.9.1 Basic definitions and properties 233
A.9.2 Well-formed subsumption relation of a graph 234
A.9.3 Consistent edge sequences and sub-paths 236
A.10 Extending Graphs with Edges 244
A.10.1 Definition and basic properties 244
A.10.2 Properties of sub-paths in an extension 245
A.11 Extending Subsomption Relations 247
A.11.1 Definition 247
A.11.2 Properties of extensions 248
A.11.3 Properties of sub-paths in an extension 249
A.12 Red-Black Graphs 256
A.12.1 Basic definitions 256
A.12.2 Extensions of red-black graphs 258
A.12.3 Building red-black graphs using extensions 261
A.12.4 Properties of red-black graphs 262
A.12.5 Relation between red-vertices 270
A.12.6 Properties about marking. 282
A.12.7 Fringe of a red-black graph 287
A.12.8 Red-black sub-paths and paths 294
A.12.9 Preservation of feasible paths 298
A.13 Conclusion 345

Résumé

Le test boîte blanche basé sur les chemins est largement utilisé pour la validation de programmes. A partir du graphe de flot de contrôle (CFG) du programme sous test, les cas de test sont générés en slectionnant des chemins d'intérêt, puis en essayant de fournir, pour chaque chemin, des valeurs d'entrées concrètes qui déclencheront l'exécution du programme le long de ce chemin.

Il existe de nombreuses manières de définir les chemins d'intérêt: les méthodes de test structurel sélectionnent des chemins remplissant un critère de couverture concernant les éléments du graphe; dans l'approche aléatoire, les chemins sont tirés selon une distribution de probabilité sur ces éléments. Ces méthodes aléatoires ont l'avantage de fournir un moyen d'évaluer la qualité d'un jeu de test à travers la probabilité minimale de couvrir un élément du critère.

Fournir des valeurs concrètes d'entrées nécessite de construire le prédicat de cheminement chaque chemin, i.e. la conjonction des contraintes sur les entrées devant être vérifiée pour que le système s'exécute le long de ce chemin. Cette construction se fait par exécution symbolique. Les données de test sont ensuite déterminées par résolution de contraintes. Si le prédicat d'un chemin est insatisfiable, le chemin est dit infaisable. Il est très courant qu'un programme présente de tels chemins et leur nombre surpassent en général de loin celui des faisables. Les chemins infaisables sélectionnés lors la première tape ne contribuent pas au jeu de test final, et doivent être tirés à nouveau. La présence de ces chemins pose un sérieux probléme aux méthodes structurelles et à toutes les méthodes d'analyse statique, la qualité des approximations qu'elles fournissent étant réduite par les données calculées le long de chemins infaisables.

De nombreuses méthodes ont été proposes pour résoudre ce problème, telles que le test concolique ou le test aléatoire basé sur les domaines d'entrée. Dans cette thèse, nous présentons un algorithme qui construit de meilleures approximations du comportement d'un programme que son CFG, produisant un nouveau CFG qui sur-approxime l'ensemble des chemins faisables mais présentant moins de chemins infaisables. C'est dans ce nouveau graphe que sont tirés les chemins.

Nous avons modélisé notre approche et prouvé formellement, à l'aide

de l'assistant de preuve interactif Isabelle/HOL, les propriétés principales établissant sa correction.

Notre algorithme se base sur l'exécution symbolique et la résolution de contraintes, permettant de détecter si certains chemins sont infaisables ou non. Nos programmes peuvent contenir des boucles, et leurs graphes des cycles. Afin d'éviter de suivre infiniment les chemins cycliques, nous étendons l'exécution symbolique avec la détection de subsomptions. Une subsomption peut être vue comme le fait qu'un certain point atteint durant l'analyse est un cas particulier d'un autre atteint précédemment: il n'est pas nécessaire d'explorer les successeurs d'un point subsumé, ils sont subsumés par les successeurs du subsumeur. Notre algorithme a été implémenté par un prototype, dont la conception suit fidèlement la formalisation, offrant un haut niveau de confiance dans sa correction.

Dans cette thèse, nous présentons les concepts théoriques sur lesquels notre approche se base, sa formalisation à l'aide d'Isabelle/HOL, les algorithmes implémentés par notre prototype, les diverses expériences menées et résultats obtenus à l'aide de ce prototype ainsi que les perspectives ouvertes par ces travaux et les améliorations qu'ils pourraient recevoir.

Abstract

White-box, path-based, testing is largely used for the validation of programs. Given the control-flow graph (CFG) of the program under test, a test suite is generated by selecting a collection of paths of interest, before providing for each path some concrete input values that will make the program follow that path during a run.

For the first step, there are various ways to define paths of interest: structural testing methods select sets of paths that fulfill coverage criteria related to elements of the graph; in random-based techniques, paths are selected according to a given distribution of probability over these elements. Both approaches can be combined as in structural statistical testing. The random-based methods above have the advantage of providing a way to assess the quality of a test set as the minimal probability of covering an element of a criterion.

The second step requires to compute, for each path, its path predicate, i.e. the conjunction of the constraints over the input parameters that must hold for the system to run along that path. This is done using symbolic execution. Then, constraint-solving is used to compute test data. If there are no input values such that the path predicate evaluates to true, the path is infeasible. It is very common for a program to have infeasible paths and such paths can largely outnumber feasible paths. Infeasible paths selected during the first step do not contribute to the test suite, and there is no better choice than to select other paths until getting feasible ones. Handling infeasible paths is the serious limitation of structural methods since most of the time is spent selecting useless paths. It is also a major challenge for many techniques in static analysis of programs, since the quality of the approximations they provide is lowered by data computed along paths that do not correspond to actual program runs.

To overcome this problem, different methods have been proposed, like concolic testing or random testing based on the input domain. In path-biased random testing, paths are drawn according to a given distribution and their feasibility is checked in a second step.

In this thesis, we present an algorithm that builds better approximations of the behavior of a program than its CFG. Our work is based on a progressive unfolding of the CFG by symbolic execution techniques and the use

of constraint solving for detecting infeasible paths. When programs contain loops, in which cases the unfolding of all paths in its CFG would yield an infinite symbolic execution tree, we introduce subsumptions to turn back this potentially infinite tree into a finite graph. A subsumption can be interpreted as the fact that some vertex met during the analysis is a particular case of another vertex met previously: there is no need to explore the successors of the subsumed vertex. An additional mechanism, the abstraction of configurations, is needed to help establishing subsumptions. The result is a transformed CFG with a finer over-approximation of the set of feasible paths, better suited for drawing paths at random.

We introduce the theoretical concepts on which our approach is based, and describe a specific graph representation and five transformations that can be combined using heuristics to compute our resulting CFG. We provide a complete formalization in Isabelle/HOL of this new graph structure and these transformations; this allows us to establish fully machine-checked proofs of the correctness of our method: actual program behaviors are preserved and no feasible path is lost by our transformations. We then present the prototype we developed to implement the method: its design closely follows the formalization, giving a good level of confidence in its correctness. This prototype implements the five operators from the model and embeds them within heuristics which can be controlled by a set of user parameters. Finally, we present the various experiments performed with our prototype and the associated results.

Chapter 1

Introduction

Random structural biased testing is a promising method of validation of systems. It is based on a graphical representation — namely, the Control Flow Graph (CFG) — of the program under test and consists in producing test cases covering paths drawn at random in the CFG, according to a given distribution. From such a path, one can automatically build a condition, the path predicate, that represents the constraints that the input values of the program must satisfy for its execution to follow that path. Test cases are produced by constraint solving from path predicates, the paths being drawn uniformly among those of a user-defined maximal length.

As for most methods of structural testing, the main problem with random structural biased testing lies in the existence of infeasible paths in the CFG of a program. Such paths do not correspond to any actual execution of the program: their presence is due to CFG being compact but approximate representations of the behaviors of programs. The existence of infeasible paths has a major negative impact on random structural biased testing since, even in simple programs, these paths tend to hugely outnumber the feasible ones. The consequence is immediate: it is usually very hard to draw feasible paths, and thus generate test cases from the CFG of the program.

In this thesis, we propose a method that, given a CFG and a precondition of the program under analysis, returns a new CFG that contains less infeasible paths. This new representation is a more accurate over-approximation of the set of feasible paths of the original program: it is a larger but more detailed version of the original graph.

1.1 Software Testing

1.1.1 Principles of Testing

Testing consists in checking that a system behaves accordingly to its intended behavior. Software testing has many interests: detecting faults in the current code, assessing its quality and the confidence in the fact that

it functions correctly. In the "Guide to the Software Engineering Body of Knowledge" [15], the authors define software testing as:

> Software testing consists of the *dynamic* verification of a programs behavior on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior.

As said above, software testing is a dynamic process: unlike static analysis techniques, it requires executing the current version of the system on a number of valuations of its inputs, the test data, observing the induced effects and interpreting those in order to decide if the system behaves as intended or not. The testing process consists of four steps: *(i)* select a set of test cases, i.e. a test suite; *(ii)* execute the system against each test case; *(iii)* decide if individual test cases failed or succeed; *(iv)* assess the quality of the test suite.

Each of these activities has its own inherent difficulties. Efficiently selecting test cases consists in finding sets of input values of the system that will allow detecting as many faults as possible. Running test cases might require to simulate a full test environment as well as the components that may miss from the system. Deciding the success or failure of a test experiment requires an oracle in order to verify the value returned by the system in that particular case and to check the state of the system after executing the program. Designing oracles for simple programs can be trivial, but it can be even more difficult than designing the system itself for complex programs, since this requires knowing for each input value the expected result, or assessing the state of the system from a finite number of external observations. Finally, being able to assess the quality of a test suite is crucial in order to decide when to stop testing the system. Each of these activities has been the subject of many works along the years, and software testing has always been a very active field of research in the domain of Computer Sciences.

1.1.2 Selecting Test Cases

In this thesis, our interest is in the selection of test cases: the goal of our work is to facilitate this part of the testing process in a specific context. Selecting test cases being a crucial part in the testing process, it has been studied by a plethora of authors over the years, and a number of different solutions already exist (see [6]). Our goal in the following is not to survey exhaustively those solutions, but to introduce a distinction that is made between test case selection techniques. Those are usually considered to be of one — or a mix of more — of the four following categories.

Selection Based on Input Domains

Test data selection based on input domains consists in choosing sets of values among the domains of the inputs of the program. Since these domains are usually very large and might even be infinite, exhaustive enumeration of the input values is not possible in most cases. The first challenge here is how to select only a finite number of test data that yield a high fault detection rate. There are globally two ways to do so: random selection of values and selection based on partitions of the input domains.

Random selection consists in selecting values of the input of the program at random according to a given probability distribution over their domains. Although easy to implement, uniform distribution usually does not lead to a high fault detection rate. This is due to the fact that, usually, a large subset of the input domain corresponds to the nominal case of execution of the program, while a number of faults require very specific combinations of input values in order to be detected; uniform distribution over large input domains might only yield a very slight chance to discover them. Another approach consists in establishing a distribution based on an operational profile (see [23, 42]): this has the advantage to facilitate detecting the faults that are the most likely to be exercised during the operation of the system. However, establishing an operational profile is a complex task. Although this method increases the level of quality experimented by the majority of the users of the program, the method does not solve the problem of detecting rare faults.

The second approach consists in partitioning the input domains in a finite number of subsets, assuming that every value of one partition has the same fault detection power: test data must cover at least each partition once. The goal is to define test data that detect classes of errors, which greatly reduces the number of test cases that must be produced. Partitioning can be done according to the type of the inputs — for example, integers could be decomposed into three classes: negative, positive integers and zero — or based on other criteria, the specification or the application domain, for example. Of particular interest are those values at the frontier, or at the vicinity, of each partition (for instance, considering values 1 or -1 in our previous example).

Specification-Based Testing

Specification-based testing takes a specification of the program as a basis for selecting test cases. A specification is a description of the functionality and constraints of the system. Its purpose is to communicate the requirements of the system to the developers. It can consist of documents written in a natural language, a collection of user-scenarios or models, a mathematical description, a prototype. Since it is not based on the code of the program

under test, specification-based testing is usually referred to as black-box testing.

Selecting test cases as early as possible in the development process, based on the specification, has a number of advantages. Since it focuses attention from the earliest stage of development on making the whole system correct, it facilitates early detection of design flaws. A carefully and early planned test strategy can also improve the reliability of the system, since this usually requires more efforts to make it to meet the requirements.

Natural language specifications are not suited to automatic generation of test cases. Usually, specification-based techniques rely on formal models, some examples of which are finite-state machines, Petri nets, systems of labeled transition systems, or specifications written in formal specification languages like OCL, JML or ACSL. In such cases, the selection of pertinent test cases can be automated, or at least partially automated.

For a survey of selection-based testing approaches, we refer the interested reader to [32].

Mutation-Based Testing

Mutation-based testing was originally conceived as a test case selection technique. However, it revealed particularly efficient in assessing the quality of test suites.

Mutation based testing, consists in generating a number of mutants, i.e. programs obtained from the program under test by injecting faults in its code. Mutants only differ from the original program for one occurrence of an operator or instruction that has been replaced by another, for example an occurrence of $\leq$ is turned into $<$. The quality of a test suite for the program is evaluated by counting the number of mutants that are killed during testing, i.e. those modified versions that are detected by one of the tests in the test suite as revealing a defect. Additional test cases must be provided to separate surviving mutants from the original program.

The practical use of mutation-based testing is limited by a number of problems. First, it usually yields large numbers of mutants, even with simple programs. This makes mutation testing an expensive method. Another problem of the approach is that it might yield mutants that, despite being syntactically different from the original programs, have the same behavior. Without surprise, such equivalent mutants cannot be killed by any test case, and it is important to be able to detect such mutants and reduce their number. Deciding the equivalence of two programs, hence of a mutant and its source, is an undecidable problem.

Since mutation-based testing can precisely evaluate the quality of test suites, it is also used to compare testing techniques.

The curious reader might refer to [36] for further details.

Structural Testing

Structural testing is based on the code of the program and, as such, is usually referred to as white-box testing. Usually, those techniques use a graphical representation of the program under test, the more common being its control flow graph (CFG). Such graphs are a compact representation of the code of the program. Their vertices represent either an elementary block of instructions or the guard of a conditional or loop statement. They also have two additional vertices that represent their unique entry and exit points. Their edges describe how the control flows between those different blocks. Structural testing consists in choosing a number of paths in the CFG. From such a path, one can compute a condition, the path predicate, that expresses the conjunction of constraints that program variables must satisfy for the execution to follow that particular path. If this condition is satisfiable, the path is said to be feasible and one can deduce a valuation of the inputs that will trigger the execution along that path, that is, a test case. If not, the path is said to be infeasible: the program can never follow that path.

Sets of paths are assessed according to a given coverage criterion: the collection of paths that will be executed during testing must cover a number of elements — or combinations of elements — of the CFG. Some of the basic coverage criteria are: statement coverage, each basic block is covered by one path at least; branch coverage, each edge must be taken at least once; decision coverage, each boolean sub-expression takes both truth values; path coverage, all paths must be selected; but there exists many other derived ones (MC/DC or data-flow based criteria, for example). This approach can be applied either with a set of paths that is initially designed to cover the criterion, or, after running an instrumented version of the code on a given test suite, to find paths that miss with the current test suite to fulfill the criterion.

Test suites might be evaluated by computing the coverage rate, i.e. the ratio of covered elements among the elements in the criterion. Achieving 100% coverage is usually very difficult and too expensive, or even impossible. For example, the explosion of the number of paths or the presence of loops might prevent full covering of the path coverage criterion. Paths that are missing for covering the criterion can also correspond to infeasible paths. The criterion can be weakened, for instance only paths of a given maximal length, or that go a given number of times through each loop, are considered, but even then, full coverage might be too expensive for complex systems. For those basic criteria, a coverage rate of 80% to 90% is usually considered good enough.

The interested reader can find more information about structural testing in [46] and [53].

1.2 Motivations

Path coverage is the most demanding criterion, since it considers that each path might carry a fault, and it is usually impossible to fully satisfy for the reasons we exposed previously. Thus, it is a natural candidate for random approaches. The distribution according to which paths are drawn must have the following properties: each path must have a non-zero probability to be selected and, pushing the idea further, this probability should be maximized, i.e. paths should be drawn uniformly. When the number of paths is infinite, because the program contains some loops, their length must be artificially bounded. How to choose this maximal length is not an easy task: it depends on the link between the test objectives and the size or structure of the graph of the program. The curious reader might refer to [27] for an analysis of the choice of length in the context of testing. How to establish an uniform distribution over paths of a maximal given length is recalled in Section 2.1.

Random structural biased testing was originally introduced by Thévenod et al. in [48] and further developed by Gouraud in [27]. It is a very efficient way to draw paths and to optimize path coverage: paths of several thousands of edges can be efficiently drawn in graphs with billions of vertices (see [20]). This random approach can also be combined with other coverage criteria, introducing a notion of coverage-guided random exploration, which in turn leads to a notion of randomized coverage satisfaction, and makes it possible to evaluate and compare different methods of random exploration according to a given criterion.

The random structural biased approach is mainly and severely limited by the existence of infeasible paths and by the fact that, even in simple programs, those paths tend to largely outnumber the feasible ones. As an example, we consider the program whose code and CFG are depicted in Figure 1.1, inspired from an industrial case study reported in [27].

This programs takes as inputs a number n of iterations of its loop to perform, and a boolean value b that artificially dictates the control flow during the execution of the loop: since the value of b does not change during the execution, paths that take both the *true* and *false* branches of the inner conditional block during different iterations of the loop are infeasible. This very simple program has no other purpose than to illustrate how the ratio of feasible paths over paths evolves as the length of these paths grows (on this example, standard optimization techniques from compiler technology would detect that the conditional can be moved outside of the loop and might emit a warning so that the developer restructure its program before sending it to test).

Since we want to select paths from which test cases can be produced, we are only interested in complete paths, i.e. paths that go from the entry point to the exit point. We refer to them as paths in the rest of this section. In the CFG of Figure 1.1b, these paths go through exactly $4k+3$ edges, where

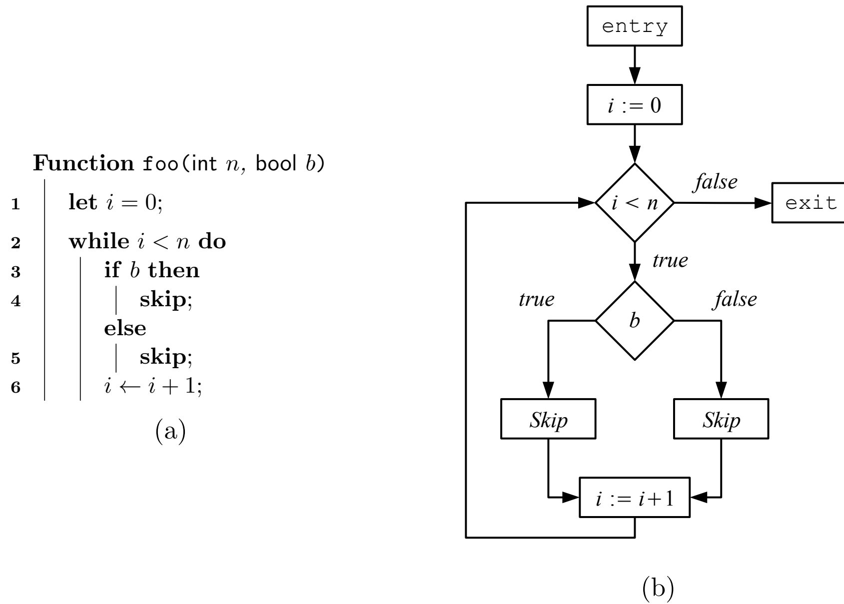


Figure 1.1: An example program (a) and its CFG (b).

k is the number of iterations of the loop each path performs. If k is zero, there exists only one path, which happens to be feasible. If $k = 1$, there are two paths and, assuming nothing limits the value of b, both are also feasible. If $k = 2$, we now have four paths, but only two of them are feasible, since the value of b now has been determined during the first iteration of the loop. There are also only two feasible paths for $k = 3$, but 8 paths, so 6 of them must be infeasible, and so on. For any $k \geq 0$, there are only 2 feasible paths but 2^k paths. Since we are interested in drawing paths of a given length at most, the probability of drawing a feasible path for a given k is:

$$\frac{1 + 2\sum_{i=1}^{k} i}{\sum_{i=0}^{k} 2^i} = \frac{1 + k(k+1)}{2^{k+1} - 1}$$

Even if this example is very simple and used for illustration purposes, it highlights the central limitation of the overall approach: the ratio of feasible paths over paths of a CFG with loops usually tend to decrease extremely fast as their length grows. As a result, it is usually very hard to draw feasible paths and thus to obtain test cases: for a large enough maximal length, drawing a feasible path in a CFG is literally looking for a needle in a haystack, but at random.

The existence of infeasible paths is not only a limitation of random structural biased testing, but of all techniques based on the CFG of the program,

or any other equivalent graphical representation. The other white-box testing techniques are also impacted — although most possibly in a lesser measure — by infeasible paths, even when considering another criterion than path coverage: since one cannot derive a test case from an infeasible path that might have been selected, another path must be chosen. Actually, the existence of infeasible paths severely impacts software testability [22]. Besides, code can be better optimized if more infeasible paths are detected during the optimization process. Infeasible path detection could also help model checking and static analysis techniques and enhance their accuracy and speed. Worst-case execution time analysis and mutation-based testing methods are also impacted by infeasible paths.

1.3 Contributions

In this thesis, we propose a method that, given a graphical representation of a program and a precondition on its inputs, produces a new graph that contains less infeasible paths. Our ultimate goal is to facilitate drawing feasible paths during the path selection phase in the context of random structural biased testing, but our approach could improve the results of other techniques based on the CFG and impacted by the existence of infeasible paths.

In full generality, infeasible path detection is an undecidable problem: a path is feasible if and only if its path predicate is satisfiable, an undecidable problem when the expressions occurring in the guards of a program are not restricted to some limited logic. As a result, we do not expect our approach to produce the graph that represent the exact set of feasible paths of any CFG, but that over-approximates it better than the CFG. Note that, even if infeasible path detection was decidable, building such a graph in each case would not be possible: the sets of feasible paths of some CFG are not regular languages, and cannot be represented by graphs. Nonetheless, experimental results reported in this document show that our approach can produce good over-approximations of these sets and lead to high infeasible path detection rates.

Our approach is based on symbolic execution of all paths of the input graph, a well-known analysis technique that consists in executing the program with symbolic inputs rather than concrete ones. This allows symbolic execution to sometimes fork at branching points, unlike concrete execution. Symbolic execution maintains the predicate of each path it follows. As a result, it can be used in conjunction with constraint solving to detect the infeasibility of some of these paths. As said previously, a major problem is that, in presence of unbounded loops, the constraints gathered along paths that go through such loops might not be sufficient to decide that the loop has to be exited at some point, preventing the analysis to terminate. Dur-

ing symbolic execution, our algorithm attempts to establish subsumptions between two occurrences of a same program location. Informally, a subsumption is established when the set of possible states of the program at a given point of the analysis is detected to be a particular case of the set of states of the program at another point met previously. The goal of finding such subsumptions is to avoid to follow cyclic paths infinitely when the program under analysis contains unbounded loops. Since it might not be possible to ever detect a subsumption, depending on the considered program and its initial set of states, our algorithm is allowed to force some subsumptions by abstracting path predicates during symbolic execution. However, forcing every possible subsumption through abstraction might only lead to detect a small number of infeasible paths, or even none. In order to improve its detection rate, our algorithm is driven by a number of heuristics that orient him towards the most accurate subsumptions during his search.

A major achievement in this thesis is the statement of a formal theory in Isabelle/HOL of an abstract version of our algorithm. This formal theory is a non-deterministic model of our algorithm, consisting of five graph transformations of a so-called red-black graph, where the red part roughly corresponds to the analyzed symbolic execution tree gained by partial unfolding of the CFG and the black part is the initial CFG of the program. Two major theoretic results were established, with fully machine-checked proofs:

1. correctness: for every path in the new graph, there exists a path with the same trace in the original one,
2. each transformation preserves the set of feasible paths.

These results apply to an entire family of algorithms that might be set on top of our current theory and combine our five transformations within specific heuristics (which node to select, which subsumption to establish, which abstraction to perform, for example), to provide approximations of the set feasible paths of a program.

The rest of this document is organized as follows. In Chapter 2, we first recall two random methods of path selection: the isomorphic and uniform random walks, the latter allowing to draw paths of a given maximal length uniformly. We then present various works that were conducted recently in the areas of infeasible path detection and symbolic execution in presence of unbounded loops.

In Chapter 3, we introduce most of the notions our approach relies on, i.e. how we represent programs, our notion of symbolic execution, subsumption, abstraction and a number of other concepts. A large part of this chapter is devoted to introduce red-black graphs and their transformations.

The formalization of our approach is described in details in Chapter 4. We present the structure of this formalization, our design choices for the

numerous and various concepts and notions it relies on, and the main lemmas that were established. The goal of this chapter is to introduce the main theorems stating the key properties of our approach as well as to give high-level descriptions of their proofs. A commented version of the full proof script is given in the appendix.

We present the prototype that implements our approach in full details in Chapter 5. This implementation is based on the formalization mentioned above, respecting a clear separation between the kernel transformations and the heuristics aspects. We also illustrate in this chapter how our algorithm behaves on a typical example, varying the different heuristics applied to illustrate how these interact and influence the construction of the result.

In Chapter 6, we first present a number of experiments that were conducted and comment and interpret the results obtained. The goal of this first part is to assess the infeasible path detection power of our approach on various examples. At the light of these experimental results, we discuss, in the second part of this chapter, the limitations of our approach in its current state as well as a number of possible improvements of our algorithm.

Finally, we conclude this thesis in Chapter 7 by summarizing and evoking some perspectives.

Chapter 2

Context: Random Testing and Infeasibility

In this chapter, we first recall in Section 2.1 the methods of isotropic and uniform random walks. Then, we present in Section 2.2 a number of works that were conducted in the area of infeasible path detection, and insist on how each of these approaches handles loops — particularly unbounded ones, since this is the aspect of the problem we are mainly concerned with. In this thesis, we chose to tackle the problem of infeasible path detection using a path based constraint propagation approach. Such approaches are based on some form of symbolic evaluation of the program under analysis. We conclude this chapter by presenting various works that tackle the problem of symbolic execution in presence of unbounded loops and from which we took a certain inspiration.

2.1 Random Structural Biased Testing

The goal of this section is to introduce the notion of uniform random walk that is used in the context of our work in order to draw paths at random in graphs. A random walk is a random exploration of a graph: from a given vertex v, one establishes a distribution of probability over the successors of v, chooses such a successor v' to visit according to this probability, and so on from v'. This requires knowing the exact structure of the explored graph.

Before introducing uniform random walks, we present the most simple of random walks, namely isotropic random walks, and explain why one should prefer the former to the latter in order to draw paths at random.

2.1.1 Isotropic Random Walks

Isotropic random walk is a classical and easy to implement method of random exploration of graphs. It consists in simply choosing the next vertex to

visit uniformly among the successors of the current vertex. In Algorithm 1, we use $succs(G, v)$ to denote the successors of a vertex v in a graph G.

Algorithm 1: The isotropic random walk algorithm.

```
input  : a graph G, a vertex v0 and a length l
output: a path p
let i = 0;
let v = v0;
let p = Nil;
while i < l ∧ succs(G, v) ≠ ∅ do
    choose a vertex v' uniformly among succs(G, v);
    append the edge (v, v') to p;
    i ⟵ i + 1;
    v ⟵ v';
return p;
```

Since it only requires knowing the successors of the current vertex, isotropic random walk seems to be the ideal candidate for exploring large graphs at random. However, the induced distribution over paths is hard to evaluate and directly depends on the topology of the explored graph. As a result, isotropic random walks might only be able to detect few faults of the program. Consider for example the graph of Figure 2.1.

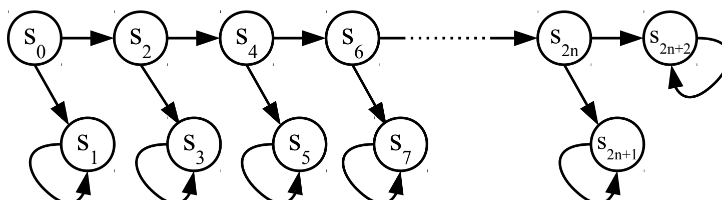


Figure 2.1: A pathological graph for isotropic random walks.

The expected number $E(N)$ of random walks needed to obtain n distinct paths of length n is:

$$E(N) = \sum_{i=0}^{n} E(N_i) = \sum_{i=0}^{n} \frac{1}{p_i} = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

where $E(N_i)$ (resp. p_i) is the expected value (resp. probability) to draw a new path after $i - 1$ different isotropic random walks. This shows that an exponential number of random walks are needed to obtain n distinct paths of a given length in the graph of Figure 2.1.

This is due to the fact that, at each visited vertex whose index is an even number (excepted for the rightmost vertex), two successors can be

chosen indistinctly: the odd successor, through which only one path goes, has the same probability to be chosen than the even one, from which an exponential number of paths start. As a result, an important number of paths are frequently shadowed by a single path and the distribution over paths is biased towards those going through odd vertices.

If the numbers of paths of a given length starting at each vertex is known at the moment it is visited, the distribution over successors can be biased to give each path the same chance to be drawn.

2.1.2 Uniform Random Walks

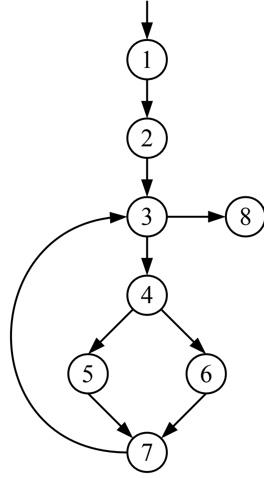
We are interested here in drawing uniformly one or several paths of length at most l in a graph G. We consider that graphs are triplets of the form (V, r, E), where V is the set of vertices, $r \in V$ an initial vertex (i.e. a root), and $E \subseteq V \times V$ a set of edges. Given a set $V_I \subseteq V$ of vertices of interest, our goal is to draw a number of paths of length at most l going from the root r to any element of V_I, but we will first focus on a simpler problem, namely drawing uniformly paths of length exactly n. We will see in the following that, by a slight modification of G, drawing paths of length at most l in G reduces to drawing paths of length l in the modified version of G.

The process is the same as for isotropic random walks: paths are built step by step, by visiting vertices and choosing which successor to visit next, adding the crossed edge to the path being built. The only difference lies in how successors are chosen: here, successors are chosen in order to give each path the same chance to be drawn. Let us suppose that vertex v is being visited, and that there remains a non-null number m of edges to cross from v in order to obtain a path of length l. The probability of each successor of v to be chosen as the next vertex to visit should be proportional to the number of paths that go through this successor. Let us note $P_v(m)$ the number of paths of length m going from any vertex v. The condition for having the uniformity over paths is to choose the successor v' to visit with probability $P_{v'}(m-1)/P_v(m)$. Computing the number $P_v(l)$ for any non negative value of l can be done using the following recurrence rules:

$$
\begin{aligned}
P_v(0) &= 1 && \text{if } v \in V_I \\
&= 0 && \text{otherwise} \\
P_v(l) &= \sum_{v' \in succs(G,v)} P_{v'}(l-1) && \text{for } l > 0
\end{aligned}
$$

As an example, we give in Figure 2.3 the underlying graph of the CFG of Figure 1.1b and its recurrence rules considering that the only vertex of interest is the exit point (denoted 8 here). The corresponding $P_v(i)$ for $0 \leq i \leq 11$ are given in Table 2.1.

The generation of n paths of length l is done in two steps:



(a)

$$
\begin{aligned}
&P_1(0) = P_2(0) = \ldots = P_7(0) = 0\\
&P_8(0) = 1\\
\\
&P_1(l) = P_2(l-1) && (l > 0)\\
&P_2(l) = P_3(l-1) && (l > 0)\\
&P_3(l) = P_4(l-1) + P_8(l-1) && (l > 0)\\
&P_4(l) = P_5(l-1) + P_6(l-1) && (l > 0)\\
&P_5(l) = P_7(l-1) && (l > 0)\\
&P_6(l) = P_7(l-1) && (l > 0)\\
&P_7(l) = P_3(l-1) && (l > 0)\\
&P_8(l) = 0 && (l > 0)
\end{aligned}
$$

(b)

Figure 2.3: The underlying graph of the CFG of Figure 1.1b (a) and its recurrence rules for $V_I = \{8\}$ (b).

1. compute $P_v(i)$, for every vertex $v \in V$ and every $0 \leq i \leq l$,

2. produce n paths of length l following the process described previously.

The first step must be done only once, whatever number of paths is to be drawn. The memory space requirements is $l \times |V|$ integer numbers; the number of arithmetic operations done during this step is $O(l \times d \times |V|)$ in the worst case scenario, where d is the maximal number of out-going edges among the vertices of G. The generation step is in $O(l \times d)$.

To draw n paths of length at most l in G, we draw n paths in the graph obtained from G by modifying it in the following way. First, we add a vertex r', which becomes the new root, to its set of vertices. Then we add an edge going from r' to itself, and an edge from r' to r. Finally, we draw n paths of length $n+1$ in this new graph. Each of these paths goes k times through the loop from r' to itself, and crosses the edge from r' to r once. By removing those $k+1$ edges from such a path, we obtain a path of length $n-k$ in G. It is straightforward to verify that any path of length at most l can produced in this manner, and that the generation is uniform.

For more details about uniform random walks and related issues, we refer the curious reader to [45].

Table 2.1: Counting table for the graph of Figure 2.2a, with $l = 11$ and $V_I = \{8\}$.

	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	$l=10$	$l=11$
$v=1$	0	0	0	1	0	0	0	2	0	0	0	4
$v=2$	0	0	1	0	0	0	2	0	0	0	4	0
$v=3$	0	1	0	0	0	2	0	0	0	4	0	0
$v=4$	0	0	0	0	2	0	0	0	4	0	0	0
$v=5$	0	0	0	1	0	0	0	2	0	0	0	4
$v=6$	0	0	0	1	0	0	0	2	0	0	0	4
$v=7$	0	0	1	0	0	0	2	0	0	0	4	0
$v=8$	1	0	0	0	0	0	0	0	0	0	0	0

2.2 Program Paths and Infeasibility

We saw previously that the existence of infeasible paths is a serious limitation not only to random structural biased testing, but to most analysis techniques based on control flow graphs. A number of works were conducted, in different contexts, in order to mitigate the loss of precision or performances caused by infeasible paths. In this section, we present some of these recent works, and insist on how they handle (unbounded) loops, since handling such loops is one the main issue when exploring the space of paths of a CFG.

A number of solutions to the infeasible path detection problem were presented in order to limit their negative impact in the data flow analysis context. Indeed, classical approaches of data flow analysis are said to be path-insensitive, i.e. they do not proceed path by path and their accuracy and performances are negatively impacted because they do not distinguish between feasible and infeasible paths.

Bodik et al. proposed in [13] an approach based on the detection of correlated branches. Given a conditional block b, one of its branch is said to exhibit some correlation if the previous statements or conditional branches taken along some path leading to b forces the execution to go through this particular branch. In the presence of such correlation, there might exist some infeasible paths because, along correlated paths, the truth value of the condition has already be determined, forcing feasible paths to go through exactly one branch. In this approach, the correlations are detected by propagating, in a backward manner by some form of reverse symbolic evaluation, the conditions of some specifically chosen branching points of the CFG. Depending on the sequence of statements the condition goes through, one might decide if its truth value at its original block was known before said block was reached. The authors do not provide a solution (or even directions) to the problem of unbounded loops: conditions are propagated at most once to each node, *de facto* bounding the number of iterations of any loop by one.

Gustafsson et al. proposed another approach in [29], with the intent of enhancing the precision of data flow analysis, in the context of worst-case execution time (WCET) analysis. These works are based on abstract execution, a variant of symbolic execution using abstract domains and operators: an infeasible path is detected when the set of possible values of at least one program variable at a given location is empty. The results obtained from this phase are then translated into flow facts that will help during WCET analysis. In this work, the authors only consider bounded loops, but do not provide solutions for the case of unbounded loops.

Another approach consists in enhancing the data flow analysis itself to make it path-sensitive (the same approach was applied to make abstract interpretation path-sensitive, see [7] for example). In [24], Fischer et al. enrich data flow analysis with predicated-lattices. Such lattice partitions the program state according to a set of predicates and tracks a lattice element for each partition. This results in predicated join operators that will only merge paths along which the same predicates hold (unlike classical data flow analysis that systematically joins paths, causing the loss of precision mentioned above). Because this analysis extends the classical data flow analysis technique, it is not impacted by unbounded loops.

The same approach, although realized in a different manner, was proposed by Das et al. in [18]. In this case, the data flow analysis is enriched with temporal logic properties, encoded as finite state machines, that are checked during the analysis. Once again, the idea is to prevent merging paths if the properties are not in the same states at join points. These works were extended by Dor et al. in [21] in order to enhance the precision of the approach for programs manipulating pointers and aliases.

As a final example, Ball and Rajamani proposed in [8] a path-sensitive version of the RHS algorithm (for Reps, Horwitz and Sagiv, see [47]) for boolean programs that uses Binary Decision Diagrams to identify infeasible paths.

Various works (including ours) about infeasible path detection use a path based constraint propagation approach. These approaches apply some form of symbolic evaluation to paths to determine their infeasibility. These methods are path-sensitive by nature: they usually have a high infeasible path detection power but with a heavy overhead. They are usually used in code optimization or software testing, in which precision is crucial.

Early works in this direction were published in 1994 in [26]. To our knowledge, this paper is one of the first to combine bounded symbolic execution of all paths — i.e. symbolic execution of all paths up to certain depth — with constraint solving in order to detect some infeasible paths. The main contribution of this paper consists in an independent constraint solver, KITP, designed specifically to detect the unsatisfiability of path predicates. Thanks to the recent advances in constraint solving, more recent works in the infeasible path detection area are less concerned in developing specific

constraint solvers, and usually rely on some efficient existing tools (like, for example, Satisfiability Modulo Theories solvers — or SMT solvers, for short — such as Alt-Ergo, CVC or Z3).

Despite these recent advances in constraint solving, a number of efforts were made to alleviate the cost of calls to solvers in path based constraint propagation approaches. Such noticeable advance is concolic testing [16, 52], where actual execution of the program under test is coupled with symbolic execution. It reduces the detection of infeasible paths to those paths that go one branch further than some feasible one, alleviating the load of the constraint solver and decreasing significantly the number of paths to be considered. This approach naturally leads to coverage of all feasible paths of a given maximal length (or that perform a pre-defined number of iterations of each loop at most).

In 1996, Altenbernd enhanced symbolic execution with WCET analysis techniques [5]. In this work, the considered programs can contain loops and calls to recursive procedures but, since those programs are all obtained from differential equations, the maximal numbers of iterations of each loop and recursive calls can all be easily determined in advance.

More recently, Bjørner and Tillman used the path based constraint propagation approach to detect infeasible paths in programs relying on string libraries [12]. The idea is to apply, given a unique path, an integer abstraction of the string constraints occurring along that path. This abstraction is then passed to a SMT solver. If this abstraction is unsatisfiable, the path is reported as infeasible. Otherwise, the solver produces a model that fixes lengths of enough strings to reduce the problem to a finite domain; the resulting fixed-length string constraints are then solved in a second step. Fixing the length of strings naturally reduces the problem to artificially bounded loops only (moreover, the search is also bounded by a maximal number of calls to the solver).

Another approach of infeasible path detection is based on the syntax of the program. Most recent and noticeable works in this area were conducted by Ngo and Tan [43]. In this paper, the authors identified four syntactic patterns that are searched, during test case selection, among the traces of selected paths: a path whose trace exhibits an instance of such a pattern is very likely (but not guaranteed) to be infeasible. The advantage of this approach comes from the fact that it avoids the expensive symbolic evaluation and constraint solving steps, since paths are never checked to be infeasible but rather considered infeasible. This gain in performance comes at the cost of a loss of precision: such syntactic techniques are prone to reporting some feasible paths as infeasible.

A recent approach of infeasible path detection, presented in [19], is based on infeasible path generalization. The underlying idea is that if a sub-path is known to be infeasible, then all paths that contain this sub-path are also infeasible. The approach is based on constraint-based explanation [41], a

technique developed in Constraint Programming to explain unsatisfiability. From an infeasible path, the approach builds a (potentially infinite) whole family of infeasible paths that will not have to be considered in the following path predicates solving phase.

Finally, we would like to mention early works that were conducted in order to estimate the number of infeasible paths of a procedure — rather than precisely identifying those paths — based on certain static code attributes. The most famous work was presented by Malevris et al. in [38]. According to the authors — and rather unsurprisingly — the greater the number of guards a path goes through, the more likely it is to be infeasible. As a result, the number of guards involved in a path makes a good metrics for assessing its feasibility. The advantage of such metrics is that they are easy to implement and provide a fast way to predict path infeasibility with a reasonable degree of confidence (see [50]), but they do not help actually detecting infeasible paths however.

2.3 Symbolic Execution and Unbounded Loops

Symbolic execution of all paths is a well-known analysis technique that consists in executing the program with symbolic inputs rather than concrete ones. Unlike concrete execution, symbolic execution might fork at branching points: this is because the symbolic values of program variables at such point might not be sufficient to decide what is the truth value of the condition guarding that point. During symbolic execution, a condition — the path predicate — is maintained along each path: it is the conjunction of the constraints that the inputs of the program must satisfy in order for its execution to follow that particular path. Constraint solving can be used in conjunction with symbolic execution to detect infeasible paths. This can be done in two manners: *(i)* on the fly, by calling a constraint solver anytime the current path might become infeasible; *(ii) a posteriori*, after all paths have been followed (if symbolic execution ever halts). The former has the advantage that infeasible paths are discovered as soon as possible, but usually requires a large number of calls to the solver, while the latter only needs to call the solver once per path, but only detects infeasible paths a posteriori. Thanks to the recent advances in constraint solving, combining symbolic execution to the former is a very precise method for detecting the infeasibility of a path or a number of paths.

Symbolic execution of all paths is limited by two major problems: the combinatorial explosion of the number of paths to consider and the presence of unbounded loops. In the latter case, due to the use of symbolic values, the constraints gathered along paths that go through such loops might not be sufficient to decide that the execution should halt at some point: such a path might be followed infinitely, preventing the analysis to stop. In the

recent years, a number of works conducted in the area of software verification proposed some solutions for these two problems. These approaches have the same goal — proving that the program is safe w.r.t. to some property — and globally proceed in the same manner: the property is encoded by an error statement in the code of the program, and the latter is proven safe if this statement is never executed in any run of the program, i.e. if all paths leading to this statement are infeasible. These approaches tackle the two previous problems by trying to detect subsumptions during the analysis. Roughly speaking, a subsumption expresses the fact that some point met during the analysis is a particular case of a point met previously. As a result, it is not needed to explore the descendants of the subsumed point: they are particular cases of the descendants of the subsumer. Since subsumptions might not be found naturally, those methods rely on some form of abstraction in order to force subsumptions. To preserve the unreachability of error statements, these abstractions are refined over time, until the program can be proved safe or unsafe (if the analysis terminates).

Our approach is directly inspired by some of these works, that we report in this section, but with a major difference. These approaches aim at detecting the unreachability of one (or several) pre-determined program locations only. Unlike our work, the goal is not to produce an over-approximation of the set of feasible paths of the program that would be as precise as possible, but one that is precise enough to prove that the program is safe w.r.t. to the considered property. In this sense, these approaches can be considered lazy w.r.t. infeasible path detection. Nonetheless, they constitute a good basis for our work since their goal is inherently the same than ours: detecting infeasible paths while over-approximating the set of feasible ones.

We took direct inspiration from Tracer [35] (and, although in a lesser measure, from the wider class of CEGAR-like systems [11, 17, 28, 33, 39] based on predicate abstraction). Tracer is a tool that combines symbolic execution of all paths, detection of subsumptions, abstraction and a counterexample guided refinement mechanism based on interpolation and weakest precondition calculus. The algorithm implemented by Tracer can be considered eager in the sense that it will always try first to produce the loosest abstraction possible to force subsumption, and lazy in the sense that it will only refine those abstractions that cause reporting false positives, i.e. that cause some infeasible path leading to an error statement to be considered feasible. During refinements, Tracer uses a mix of weakest precondition calculus and interpolation to produce conditions that capture the reason of the infeasibility of a given path from a given program location. Such conditions represent a subset of the constraints that all program states must satisfy at this point. These conditions serve two purposes: preventing too crude abstractions in the future by forcing loops to be unfolded again, and allowing to merge different paths during the analysis, thus mitigating the explosion of paths. The purpose of interpolation here is to remove from

these conditions the facts that are irrelevant w.r.t. to the unreachability of the error statement (typically, constraints that are related to the unreachability of other statements). This yields conditions that focus on the relevant facts, which simplify the analysis but also participates in its laziness when it comes to infeasible path detection. In the following, we refer to such conditions as safeguard conditions, since their goal is to block abstractions that would prevent detecting the unreachability of some locations.

When studying Tracer in order to adapt it to our purpose, we found that the presented proof sketches in the accompanying literature revealed a sensible gap to a formal development. Approaches similar to Tracer and ours rely on a number of features and heuristics that interact in a very intricate manner, which makes them perfect candidates for machine aided formalization and pushed us toward stating the formal theory introduced in Chapter 4. To our knowledge, such formalization was never done before.

Tracer's approach is itself directly inspired from the works of McMillan related in [39] and [40]. The major difference between McMillan's and Tracer's approach lies in the way safeguard conditions are computed. In the former, weakest precondition is never used: safeguard conditions are computed by interpolation only, from the proofs of the infeasibility of paths leading to the error statements. Despite a number of minor distinctions between the two approaches — McMillan refers to the notion of subsumption as coverage, and checks for the infeasibility of paths only when those reach the end of the program — we do not observe other differences in both approaches. It is reported in [39] that the approach does not guarantee termination of the analysis in all cases (which is also the case for Tracer).

The works presented in [40] rely on the same ideas but extends the approach in order to force convergence of unbounded loops in more cases than previously, besides adapting it for programs with multiple procedures. The idea here is to instrument the program with additional variables — one for each loop — whose values decrease when loops are unfolded and increase during backtracking. In the first step, the analysis starts with concrete values for each additional variables, which actually forces each loop to be unfolded that number of times at most during the analysis. This first step of the analysis is a form of bounded model-checking, and safeguard conditions produced during this step usually contains facts that depends on the values of the additional variables. The second step of the analysis consists in purging the safeguard conditions from the facts relying on the additional variables, as well as any non-inductive constraints. This yields inductive formulae that might preserve or not the unreachability of the error nodes. If this is the case, the program is declared safe. If not, the analysis can be restarted while increasing the initial values of the additional variables. As for the works introduced in [39] this approach does not guarantee convergence in all cases, but leads to it in more cases than previously.

In all these approaches, the process of computing abstractions at the

different program locations can be seen as a form of invariant synthesis. Note however that these invariants are usually weak (they are far from being the strongest possible invariants; computing such invariants is not the goal of the approaches presented in this section) in the sense that they express some properties of the program related to its termination rather than to its functional aspects. As a result, we did not bring a particular interest to invariant synthesis — in its widely accepted meaning — in this thesis, but we do not exclude that injecting such invariants (user-provided or automatically inferred) in our analysis might improve the infeasible path detection power of our approach.

Chapter 3

Introducing Red-Black Graphs and their Transformations

3.1 Introduction

In this chapter, we introduce the notations and formalize the various concepts we will rely upon in the rest of this thesis.

We first show in Section 3.2 how we model programs in a standard way, using *Labeled Transition Systems* (LTS).

Then, we show in Section 3.3 how we model the execution of programs by *symbolic execution*. Symbolic execution is a program analysis technique that executes programs with symbolic inputs rather than concrete ones. Symbolic execution follows each execution path of the program under analysis: when a branching point is met, it forks in order to follow both paths, unlike concrete execution. Along each path, symbolic execution maintains a condition, called the *path predicate*, which is updated whenever a non-trivial instruction is met, to represent the constraints the inputs must satisfy for the execution to reach a particular program point. We say that symbolic execution performs over *configurations*, which represent sets of *program states* at a given program point. We first formalize the concept of configurations in 3.3.1, show how configurations are built using symbolic execution in 3.3.2 and finally show how symbolic execution performs on programs in 3.3.3.

In Section 3.4, we introduce the notions of *subsumption* and *abstraction of a configuration*. Subsumption is a relation between configurations: informally, we say that a configuration is subsumed by another if the former is a particular case of the latter. In our algorithm, the main problem is to detect subsumptions between configurations along paths followed by symbolic execution, to avoid unfolding loops ad infinitum. However, given two

configurations, it is usually not the case that one is a particular case of the other and subsumption cannot occur. In order to increase the likelihood of subsumption occurring, our algorithm is allowed to *abstract* configurations during the analysis. Abstracting a configuration can be seen as enlarging the set of program states it represents, which is done by forgetting part of the information the configuration carries. However, this loss of information might restrain to detect the infeasibility of some paths in the LTS of the program under analysis, thus introducing infeasible paths in the new LTS. To prevent performing too crude abstraction, configurations can be labeled by conditions acting as safeguards against such abstractions. These conditions could be provided by the user before the start of the analysis, or computed during the analysis by some kind of counterexample guided refinement, learning from faulty abstractions. We formalize subsumption in 3.4.1. We introduce abstraction and how too crude abstractions can be prevented in 3.4.2.

Then, we introduce in Section 3.5 the notion of *red-black graphs.* In the rest of this thesis, the interest of red-black graphs is twofold. First, they are a convenient data structure for stating and proving the two key properties our approach must maintain: all feasible paths of the original model are preserved, and all paths of the new model must perform the same computations than their equivalent in the original model. Second, to stay as close as possible to the formalization, our algorithm maintains a red-black graph during its execution. This red-black graph is then turned into a LTS when the analysis is over.

Finally, we introduce in Section 3.6 the various red-black graph transformations our algorithm is allowed to perform.

3.2 Modeling programs

Programs are modeled by Labeled Transition Systems:

Definition 1. A *Labeled Transition System* (LTS) is a quadruple (L, l_i, Δ, F) where:

- L is a finite set of program locations,
- $l_i \in L$ is the initial location, i.e. the unique entry point of the program,
- $\Delta \subseteq L \times \mathit{Labels} \times L$ is a finite set of transitions, where *Labels* is a set of labels whose elements represent the basic operations that can occur in programs,
- $F \subseteq L$ a set of final locations, i.e. exit points.

We note *Vars* the set of program variables and we represent labels by the

following datatype:

$$label \ ::= \ \text{Skip} \mid \text{Assume } \phi \mid \text{Assign } v \ e$$

where ϕ is a boolean expression over the elements of $Vars$, v is a program variable and e an arithmetic expression over the elements of $Vars$.

We suppose LTS to be equipped with the following applications: $src : \Delta \rightarrow L$, $tgt : \Delta \rightarrow L$, $label : \Delta \rightarrow Labels$ and $trace : \Delta^* \rightarrow Labels^*$, which respectively associate their source, target and label to transitions and their trace to sequences of transitions. Given a location l of an LTS, we note $\Delta_i(l)$ and $\Delta_o(l)$ the sets of in-going and out-going transitions of l, respectively.

Location l_i has no incoming transition and elements in F have no out-going transitions. Each location l of L is reachable from l_i, and there exists a final location in F reachable from l. The transition relation represents the operations that are executed when control flows from a program location to another. We write $l \overset{label}{\rightarrow} l'$ to denote the transition leading from $l \in L$ to $l' \in L$ executing the operation corresponding to $label \in Labels$. Conditional statements are directly encoded using the underlying graph structure of the LTS by adding transitions labeled with Assume ϕ or Assume $\neg\ \phi$ to the successors.

Such LTS model programs as if they were the result of a pre-compiler for a simple imperative programming language where basic operations are either assignments or *Skip*. *Skip* is used for transitions associated with statements controlling the flow of execution, like `break`, `continue` or `goto` statements. Conditional statements are either If-Then-Else blocks (the Else-branch being optional) or While-loops.[1] Conditionals are assumed to have no side-effects. There is no explicit block structure: it is assumed that, after some preliminary scope analysis and renaming, all variables are defined at the topmost level. We also suppose that this scope analysis ensures that variables used without being defined are all inputs of the program (global variables or formal parameters), not uninitialized local variables. Given two distinct locations l and l' of an LTS, we suppose that there can be only one transition going from l to l'. Finally, we consider only "loop free" LTS in which no vertex has an edge on itself.

Since CFG have usually a unique exit point, we could have define LTS with a unique final location f in place of the set F. However, our approach produces, given an LTS and an initial configuration of the program, a new LTS in which locations are occurrences of the locations of the input LTS. Thus, our results might contain several occurrences of the unique original final location. The previous definition covers both cases of LTS.

The domain of all program variables is noted D. A *program state* is a valuation of program variables.

[1] When the Else-branch is missing, we add to the LTS one transition that is: *(i)* labeled by the negated condition for the missing branch; *(ii)* pointing to the vertex at the end of the block.

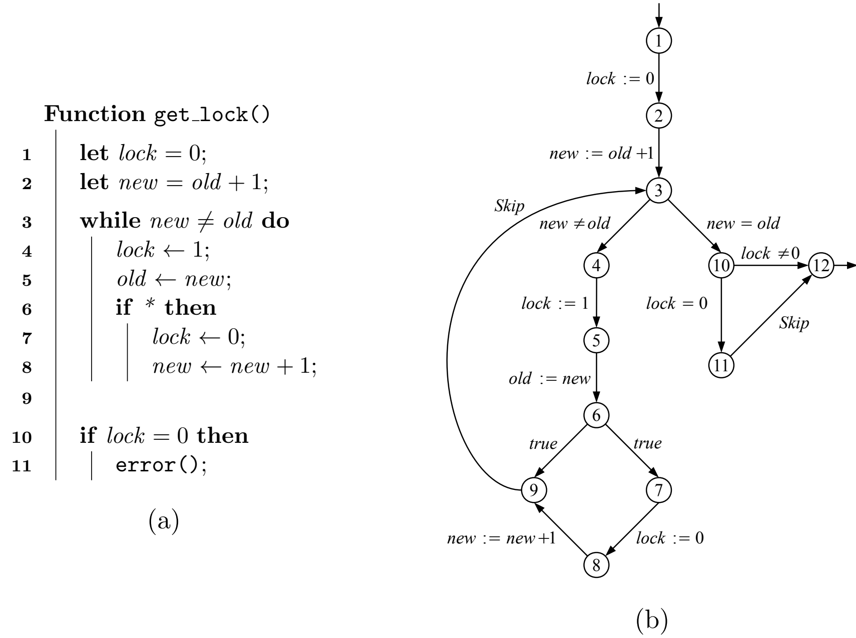


Figure 3.1: A lock acquisition program (a) and its LTS (b).

Definition 2. A *program state* $\sigma : Vars \to D$ is a function associating values to program variables.

In the examples, we use $D = \mathbb{N}$. This could be extended to arrays, records and other constructions with standard restrictions: the only limitation is the one given by the constraint solver in use. Restrictions linked to the logic supported by constraint solvers or theorem-provers are quite classic in this field of study.

Example 1. The program in Figure 3.1a (borrowed from [31]) implements a lock acquisition algorithm. The process or thread executing this program enters the loop (line 3) and first acquires the lock (line 4), but releases it immediately if the call to an external condition - like a function or system call - succeeds (line 6). The condition $*$ of the **if** statement at line 6 abstracts the call to the external condition. Since nothing is known about this condition, it can be indistinctly *true* or *false* whenever it is called. The program has been instrumented to prove that, when the execution exits the loop, *lock* must be 1. If this is not the case, an error is raised (line 11). In [31], proving the partial correctness of this code is viewed as proving that any path leading to line 11 is infeasible.

The LTS of this program is given in Figure 3.1b. For readability reasons, transitions are labeled with corresponding program statements. Since the condition `*` at line 6 can be either *true* or *false* anytime it is used and

since the result of the call is not stored, both branches of this conditional statement have been translated into transitions labeled by Assume *true*. Moreover, since: *(i)* our interest is not in verifying programs, but pruning infeasible paths from their LTS representation, *(ii)* our input language does not include an `error()` statement, the instruction `error()` at line 11 has been translated into a Skip label. By building a new LTS, which includes all feasible paths of the original CFG and performs the same computations, in which the edge from 10 to 11 is marked as infeasible, our approach is also able to show that the function will not exit with $lock = 0$. However, our objective is to prune infeasible paths in general, not only the ones associated with program correctness. □

3.3 Operational Semantics of Programs

3.3.1 Configurations

Symbolic execution consists in executing programs giving symbolic values to their inputs rather than concrete ones. Symbolic values are represented by so called *symbolic variables* that we model by indexed version of program variables. The set $Vars \times \mathbb{N}$ of all symbolic variables is denoted $SymVars$.

Notation: to ease the reading we note v_i the symbolic variable (v, i).

At each step of symbolic execution, one must keep track of two types of facts:

- the symbolic values associated to program variables,
- the constraints that these symbolic values must satisfy for the execution to reach the current program point.

We formalize these notions by the concept of *configuration.*

Definition 3. A *configuration* is a pair (s, π) where:

- $s : Vars \to \mathbb{N}$, called the *store*, is a function from program variables to indexes,
- π, called the *path predicate*, is a formula over symbolic variables.

We use stores to map program variables to symbolic variables. Given a store s and a program variable v, the symbolic variable mapped to v by s is the pair $\big(v, s(v)\big)$.

The path predicate is a conjunction: it records the constraints that the different symbolic values associated to program variables have to satisfy in order for an actual execution to reach the corresponding program point.

Example 2. Let c_1 be the configuration:

$$\big(\{lock \mapsto 0, new \mapsto 0, old \mapsto 0\},\ true\big)$$

It maps program *lock* (resp. *new* and *old*) to symbolic variable $lock_0$ (resp. new_0 and old_0). Its path predicate is simply *true*, i.e. nothing is known about the symbolic values of the three program variables when entering the corresponding program point.

The configuration:

$$c_3 = \big(\{lock \mapsto 1, new \mapsto 1, old \mapsto 0\},\ lock_1 = 0 \wedge new_1 = old_0 + 1\big)$$

would be computed during symbolic execution of the program given in Figure 3.1a with initial configuration c_1 when reaching the first occurrence of program location 3, i.e. by symbolic execution of transitions $(1, \text{Assign } lock\ 0, 2)$ and $(2, \text{Assign } new\ old + 1, 3)$, in this order. It maps the program variable *lock* to the symbolic variable $lock_1$, *new* to new_1, etc. Its path predicate states that the symbolic value of *lock* must be zero (suppose that the domain is the set of integers) and the symbolic value of *new* must equals the symbolic value of *old* plus one. Nothing is known about the symbolic value of old_0, hence about the value of program variable *old*. $\square$

The set of all configurations is denoted by $\mathfrak{C}$. A configuration represents a set of program states. We define the concepts needed to formalize this notion of program states represented by a configuration.

Similarly to a program state, a *symbolic state* is a function from symbolic variables to values.

Definition 4. A *symbolic state* $\sigma_{sym} : SymVars \to D$ is a function associating values to symbolic variables.

Given a store s, a program state and a symbolic state are said to be *consistent with s* if, for each program variable v, they associate the same value to v and to the symbolic variable it is mapped to by s.

Definition 5. Let $s : Vars \to \mathbb{N}$ be a store, $\sigma : Vars \to D$ a program state and $\sigma_{sym} : SymVars \to D$ a symbolic state. σ and σ_{sym} are *consistent with* s, noted $cons(\sigma, \sigma_{sym}, s)$, if :

$$\forall v \in Vars.\ \sigma\ v = \sigma_{sym}\ \big(v, s(v)\big)$$

Given an arithmetic or boolean expression e over program (resp. symbolic) variables and a program state σ (resp. a symbolic state σ_{sym}), we write $e(\sigma)$ (resp. $e(\sigma_{sym})$) the evaluation of e in σ (resp. σ_{sym}).

The *set of program states represented by a configuration* c, or simply the set of states of c, is the set of program states σ for which there exists a symbolic state σ_{sym} such that: *(i)* σ and σ_{sym} are consistent with the store of c; *(ii)* σ_{sym} satisfies the path predicate of c.

Definition 6. Let $c = (s, \pi)$ be a configuration. The *set of states of* c, noted $\mathit{States}(c)$, is the set $\{\sigma.\ \exists \sigma_{sym}.\ \mathit{cons}(\sigma, \sigma_{sym}, s) \wedge \pi(\sigma_{sym})\}$.

A configuration $c = (s, \pi)$ is *satisfiable* if there exists a symbolic state that satisfies its path predicate π. This is equivalent to say that its set of states is not empty.

Definition 7. A configuration $c = (s, \pi)$ is *satisfiable* if there exists a symbolic state σ_{sym} such that $\pi(\sigma_{sym})$ holds.

Lemma 1. *A configuration c is satisfiable if and only if $\mathit{States}(c) \neq \emptyset$.*

Proof. Suppose that $c = (s, \pi)$ is satisfiable and let σ_{sym} be a symbolic state that satisfies π. By Definition 5, the program state that associates $\sigma_{sym}(v, s(v))$ to all program variables is consistent with s. Thus, σ is a state of c, by Definition 6.

Suppose that $\mathit{States}(c)$ is not empty and let σ be a state of c. By Definition 6 there exists a symbolic state σ_{sym} such that σ and σ_{sym} are consistent with s and which satisfies π, thus c is satisfiable. □

Example 3. The configuration:

$$c_3 = (\{\mathit{lock} \mapsto 1, \mathit{new} \mapsto 1, \mathit{old} \mapsto 0\},\ \mathit{lock}_1 = 0 \wedge \mathit{new}_1 = \mathit{old}_0 + 1)$$

given in Example 2 is satisfiable. For example, the symbolic state σ_{sym} which associates 0 to lock_1, 1 to new_1 and 0 to old_0 is a model of its path predicate. Moreover, its set of states is not empty, as one can build from σ_{sym} a program state σ such that both are consistent with the store of c_3. This σ would associate 0 to *lock*, 1 to *new*, etc.

Let c_{10} be the following configuration:

$$(\{\mathit{lock} \mapsto 1, \mathit{new} \mapsto 1, \mathit{old} \mapsto 0\}, \mathit{lock}_1 = 0 \wedge \mathit{new}_1 = \mathit{old}_0 + 1 \wedge \mathit{new}_1 = \mathit{old}_0)$$

which would be computed when reaching the first occurrence of location 10 during symbolic execution of the program depicted in Figure 3.1a, i.e. by

symbolic execution of transitions $(1, \text{Assign } \mathit{lock}\ 0, 2)$, $(2, \text{Assign } \mathit{new}\ \mathit{old} + 1, 3)$ and $(3, \text{Assume } \mathit{new} = \mathit{old}, 10)$, in this order. This configuration is unsatisfiable, since its path predicate requires new_1 to be both equal and greater than old_1. Its set of states is empty, hence subpath $1 \cdot 2 \cdot 3 \cdot 10$ is infeasible. □

Finally, we say that a configuration c entails a boolean expression over program variables if this expression evaluates to *true* for all states of c.

Definition 8. Let c be a configuration and ϕ a boolean expression over program variables. We say that c *entails* ϕ if $\forall \sigma \in \mathit{States}(c).\ \phi\ \sigma$.

3.3.2 Symbolic Execution Steps

We represent symbolic execution as a function *SE*, which takes a configuration and a label as inputs and produces a second configuration.

Definition 9. *Symbolic execution* is a function *SE* from $\mathfrak{C} \times \mathit{Labels}$ to $\mathfrak{C}$ defined as follows :

$$\mathit{SE}\ c\ l = \begin{cases} c & \text{if } l = \text{Skip} \\ \big(s,\ \pi \wedge [\![\phi]\!]_s\big) & \text{if } l = \text{Assume } \phi \\ \big(s(v := i),\ \pi \wedge (v, i) = [\![e]\!]_s\big) & \text{if } l = \text{Assign } v\ e \end{cases}$$

where:

- $[\![e]\!]_s$ (resp. $[\![\phi]\!]_s$), called the *adaptation of* e *(resp.* ϕ*) to the store* s, denotes the expression obtained from e (resp. ϕ) by substituting every occurrence of any program variable v by the symbolic variable $\big(v, s(v)\big)$,
- $f(x := y)$ denotes the function associating y to x and $f(z)$ for all $z \neq x$, i.e. $s(v := i)$ is the store obtained from s by associating a new index i to v. The new index i must be such that the symbolic variable (v, i) is fresh for c, i.e. it is not yet associated to a program variable by the store and it does not occur in the path predicate.

Example 4. Symbolic execution of label Assign *lock* 0 from the initial configuration c_1 given in Example 2 gives the configuration c_2:

$$\begin{aligned} c_2 &= \mathit{SE}\ c_1\ (\text{Assign } \mathit{lock}\ 0) \\ &= \big(\{\mathit{lock} \mapsto 1, \mathit{new} \mapsto 0, \mathit{old} \mapsto 0\},\ \mathit{lock}_1 = 0\big) \end{aligned}$$

Its store associates the index 1 to program variable *lock*, and the constraint $\mathit{lock}_1 = 0$ relating the program variable *lock* to its symbolic value (which happens to be concrete) has been added to the path predicate.

Symbolic execution of label Assign $new\ old + 1$ from c_2 yields the configuration c_3:

$$\begin{aligned} c_3 &= SE\ c_2\ (\text{Assign } new\ old + 1) \\ &= \big(\{lock \mapsto 1, new \mapsto 1, old \mapsto 0\},\ lock_1 = 0\ \wedge\ new_1 = old_0 + 1\big) \end{aligned}$$

Its store has been updated for program variable new. The right-hand side of the new constraint $new_1 = old_0 + 1$ has been obtained by substituting the only occurrence of old in $old + 1$ by its symbolic counterpart at c_2.

Symbolic execution of label Assume $new \neq old$ from c_3 gives the configuration:

$$\begin{aligned} c_4 &= SE\ c_3\ (\text{Assume } new \neq old) \\ &= \big(\{lock \mapsto 1, new \mapsto 1, old \mapsto 0\}, \\ &\qquad lock_1 = 0\ \wedge\ new_1 = old_0 + 1\ \wedge\ new_1 \neq old_0\big) \end{aligned}$$

Again, the new conjunct in the path predicate has been obtained by substituting occurrences of program variables in the guard $new \neq old$ by the symbolic variables given by c_3. □

In the case where l is an assignment, the way the new index i is chosen has no real influence on the result of SE as long as (v, i) is fresh for c: different choices for i yield configurations with identical sets of states. In practice, we just take the successor of the current index, as shown in the previous example.

Lemma 2. *Let l be a label and c_1 and c_2 two configurations such that $States(c_1) = States(c_2)$, then*

$$States(SE\ c_1\ l) = States(SE\ c_2\ l)$$

Proof. The proof is obtained by structural induction on l:

- it trivially follows Definition 9 when $l = \text{Skip}$,
- given a configuration c and a label $l = \text{Assume } \phi$, one can show that

$$States(SE\ c\ l) = \big\{\sigma \in States(c).\ \phi(\sigma)\big\} \tag{3.1}$$

  Thus

$$\begin{aligned} States(SE\ c_1\ l) &= \big\{\sigma \in States(c_1).\ \phi(\sigma)\big\} \\ &= \big\{\sigma \in States(c_2).\ \phi(\sigma)\big\} \\ &= States(SE\ c_2\ l) \end{aligned}$$

- similarly, given a configuration c and a label $l = \text{Assign } v\ e$, one can show that

$$\mathit{States}(\mathit{SE}\ c\ l) = \big\{\sigma(v := e(\sigma)) \mid \sigma.\ \sigma \in \mathit{States}(c)\big\} \qquad (3.2)$$

  Thus

$$\begin{aligned}\mathit{States}(\mathit{SE}\ c_1\ l) &= \big\{\sigma(v := e(\sigma)) \mid \sigma.\ \sigma \in \mathit{States}(c_1)\big\}\\ &= \big\{\sigma(v := e(\sigma)) \mid \sigma.\ \sigma \in \mathit{States}(c_2)\big\}\\ &= \mathit{States}(\mathit{SE}\ c_2\ l)\end{aligned}$$

□

Proof for equation (3.1) directly follows Definitions 5, 6 and 9. Proving equation (3.2) requires a great number of technical details so we skip its proof here, but it is given (as well as the corresponding lemma) in Appendix A.6.3.

Symbolic execution from an unsatisfiable configuration always yields an unsatisfiable configuration. This property of *SE* is in accordance with the fact that infeasible paths are made of a (potentially empty) feasible prefix and a (non-empty) infeasible suffix.

We extend *SE* to lists of labels, which is realized by the following function SE^*. Properties of *SE* also hold for SE^*.

Definition 10. SE^* is a function which takes a configuration and a list of labels as inputs and returns a configuration, defined as follows

$$\mathit{SE}^*\ c\ \mathit{ls} = \begin{cases} c & \text{if } \mathit{ls} \text{ is empty}\\ \mathit{SE}^*\ \big(\mathit{SE}\ c\ (\mathit{hd}\ l)\big)\ (\mathit{tl}\ \mathit{ls}) & \text{otherwise}\end{cases}$$

where *hd* and *tl* give the head and the tail of a list, respectively.

A list of labels is said to be *feasible from a configuration c* if its symbolic execution from c yields a satisfiable configuration.

Definition 11. A list of labels *ls* is *feasible* from a configuration c if $SE^*\ c\ ls$ is satisfiable.

3.3.3 Symbolic Execution of Programs

Symbolic execution of a program is performed over its LTS representation, given an initial configuration whose store associates 0 to every program variable. In general, the path predicate of this initial configuration is simply *true*, however nothing restrains from starting the analysis with a different path predicate which would encode a given precondition.

Symbolic execution follows every path of the LTS representation of the program and maintains a configuration which is updated, using *SE* as defined in the previous subsection, whenever a transition is taken. When a branching point is met, symbolic execution *forks* and continues along both paths, unlike concrete execution. Whenever the current configuration is proved unsatisfiable, thanks to a call to a constraint solver, symbolic execution halts along the current path: there exists no valuation of the input variables such that the execution reaches the current program point. Since the current configuration has been proved unsatisfiable, its successors are also unsatisfiable. The current path is said to be infeasible.

Paths followed during symbolic execution can be represented by a *symbolic execution tree*. Its nodes and leaves are occurrences of locations of the original program; they are labeled by the configurations computed at the corresponding program points. Its branches are labeled by the corresponding labels in the original LTS. In presence of loops, and without a specific treatment for them, one obtains a potentially infinite symbolic execution tree. Adding constraint-solving to symbolic execution allows to detect some infeasible paths, sometimes a fair number of them, while following all feasible paths. In general, symbolic execution methods result in an over-approximation of the set of feasible paths.

In software testing, symbolic execution is used to generate test inputs for each path of the symbolic execution tree that was not detected infeasible. For example, given an initial configuration whose path predicate is *true*, symbolic execution of the simple program in Figure 3.2a yields the symbolic execution tree depicted in Figure 3.2c. It has four paths: two feasible and two infeasible, the latter having their last step marked by a $\perp$ symbol in Figure 3.2c. The two feasible paths can be executed by running the program on the inputs $\{a = 3, b = 1\}$ and $\{a = -2, b = 0\}$. As an example, we give the configurations computed at the four leaves of this symbolic execution tree. These four configurations have the same store, since c is assigned twice on each path: $\{a \mapsto 0, b \mapsto 0, c \mapsto 2\}$. Their path predicates are:

- $\pi_{6^0} \;\equiv\; c_1 = a_0 \wedge c_1 < b_0 \wedge c_2 = b_0 - c_1 \wedge c_2 < 0,$
- $\pi_{7^0} \;\equiv\; c_1 = a_0 \wedge c_1 < b_0 \wedge c_2 = b_0 - c_1 \wedge c_2 \geq 0,$
- $\pi_{6^1} \;\equiv\; c_1 = a_0 \wedge c_1 \geq b_0 \wedge c_2 = c_1 - b_0 \wedge c_2 < 0,$
- $\pi_{7^1} \;\equiv\; c_1 = a_0 \wedge c_1 \geq b_0 \wedge c_2 = c_1 - b_0 \wedge c_2 \geq 0$

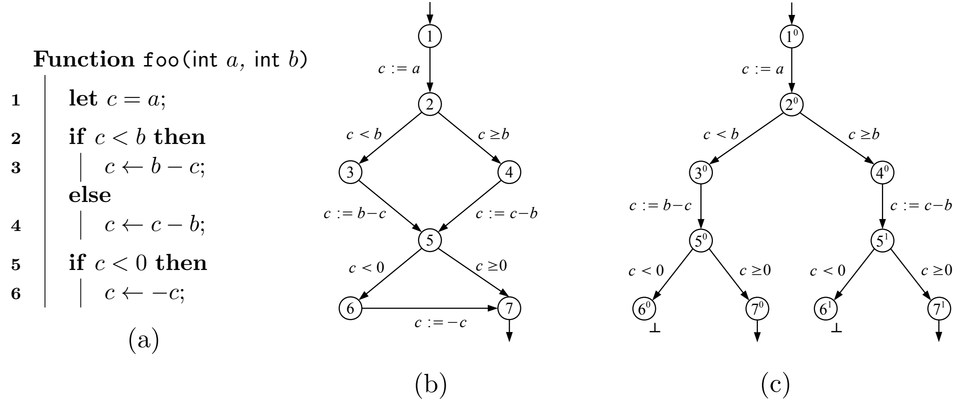


Figure 3.2: A simple example program (a), its LTS (b) and its symbolic execution tree (c). Configurations are not shown.

with π_{6^0} and π_{6^1} being unsatisfiable.

Symbolic execution can also be used for verification purposes [34, 40] For example, the program in Figure 3.2a computes in c the distance between its inputs a and b. When it ends, c must be greater or equal to 0. As shown in Figure 3.2c, paths leading to an occurrence of program location 6 are detected infeasible thus no run of the program can execute the last assignment.

Besides restrictions due to the use of constraint-solvers or theorem-provers for checking the satisfiability of path predicates, symbolic execution is also limited by the unbounded loops in the source code. In their presence, symbolic execution might unfold indefinitely the same program path, resulting in an infinite symbolic execution tree. For example, symbolic execution of the lock acquisition program in Figure 3.1a from the initial configuration $c_1 = \big(\{lock \mapsto 0, new \mapsto 0, old \mapsto 0,\ \ true\}\big)$ given in Example 2 yields the infinite symbolic execution tree depicted in Figure 3.3b. The path that does any number of iterations of the loop but that never goes through transition $(6, \text{Assume } true, 9)$ is feasible and can always be extended by another such iteration.

A classical way to mitigate this problem is to force symbolic execution to end by adding some timeout conditions, for example to stop following a path when it reaches a given maximal length [25] or when all loops have been iterated at most a given number of times along each path [52]. However, this solution is obviously not ideal, since in this case the resulting symbolic execution tree only represents a finite set of prefixes of feasible paths. This can prevent to discover faults that might be only observable beyond these time out conditions.

In the next section, we introduce an elaborated solution based on subsumption and abstraction to address this problem. The idea is to force

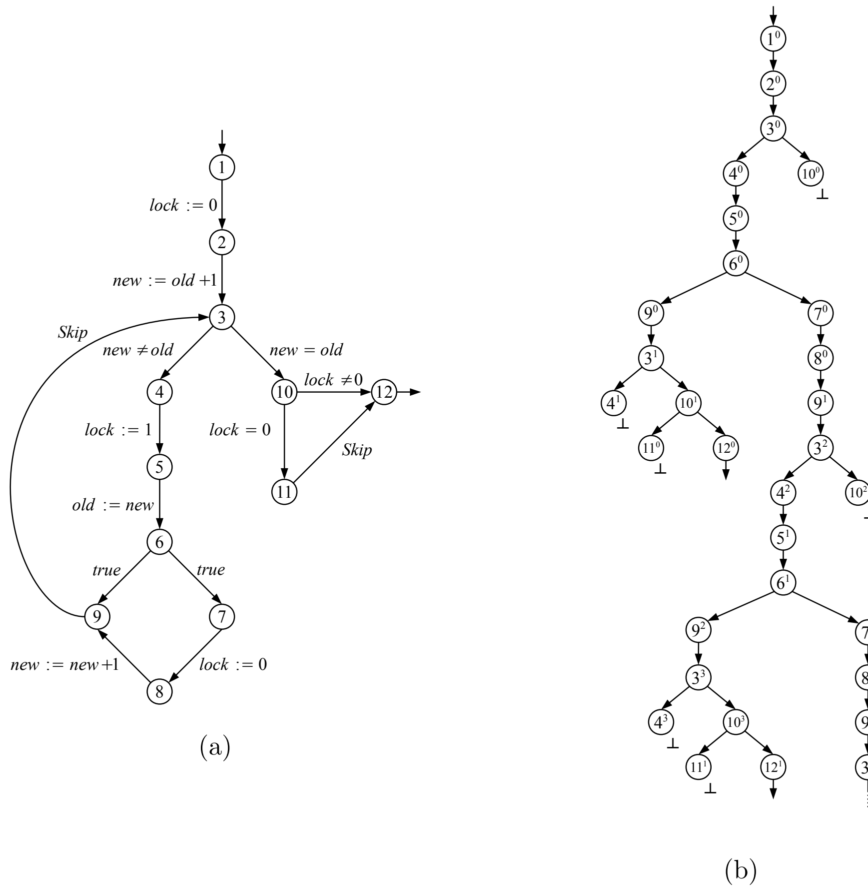


Figure 3.3: The LTS (a) and the symbolic execution tree (b) of the lock acquisition program given in Figure 3.1a: the feasible path that never goes into the inner conditional does not end.

stopping the unfolding of loops by detecting subsumptions between the vertices of the symbolic execution tree, possibly forcing them with abstraction. This will turn the potentially infinite symbolic execution tree into a complete finite symbolic execution graph which over-approximates the set of feasible paths of the original LTS.

3.4 Subsumption

3.4.1 Subsumption

In order to avoid unfolding unbounded loops infinitely, we enrich symbolic execution with the detection of *subsumptions*. As said earlier, subsumption is a relation between configurations: roughly speaking, a configuration c' is subsumed by a configuration c if it is a particular case of c. More precisely,

c' is subsumed by c if its set of states is a subset of the set of states of c.

Definition 12. Let c and c' be two configurations: c' is *subsumed* by c, noted $c' \sqsubseteq c$, if $States(c') \subseteq States(c)$.

In practice, we will only be interested in subsumptions taking place between occurrences of the same program location. Moreover, these program locations must represent loop headers. When following an execution path, every time a loop header is reached, the algorithm checks if a subsumption can apply with one of the previous occurrences of the same loop header. If the subsumption is established, symbolic execution of the current path halts: successors of a subsumed program point are subsumed by successors of the subsumee. If not, symbolic execution continues.

Example 5. The configuration:

$$c_{3^0} = \big(\{lock \mapsto 1, new \mapsto 1, old \mapsto 0\},\ lock_1 = 0\ \wedge\ new_1 = old_0 + 1\big)$$

computed for 3^0, the first occurrence of program location 3, subsumes the configuration at point 3^2 :

$$\begin{aligned} c_{3^2} = \big(&\{lock \mapsto 3, new \mapsto 2, old \mapsto 1\},\ lock_1 = 0\ \wedge\ new_1 = old_0 + 1\ \wedge \\ &lock_2 = 1\ \wedge\ old_1 = new_1\ \wedge\ lock_3 = 0\ \wedge\ new_2 = new_1 + 1\big) \end{aligned}$$

Indeed, both configurations require *lock* to be 0 and *new* to be *old* plus one.

However, it does not subsume the configuration:

$$\begin{aligned} c_{3^1} = \big(&\{lock \mapsto 2, new \mapsto 1, old \mapsto 1\},\ lock_1 = 0\ \wedge\ new_1 = old_0 + 1\ \wedge \\ &lock_2 = 1\ \wedge\ old_1 = new_1\big) \end{aligned}$$

computed for program point 3^1 since, unlike c_{3^0}, c_{3^1} requires *lock* to be 1 and *new* and *old* to be equals. □

Subsumption between two occurrences of a loop header corresponds to the inclusion of the set of states of the subsumee into the set of states of the subsumers, and this inclusion is based on the path predicates of both configurations. If the path predicate of the possible subsumer includes few constraints a subsumption can occur without being always meaningful. Such an example is a program whose first instruction is a loop. In absence of any user precondition, the path predicate for the first visit of the header of the loop is *true* and the corresponding configuration can subsume any further occurrence of that loop header along a symbolic path[2]. We need additional

[2]This situation can also happen as the result of successive abstractions of a configuration, as explained in the next section.

heuristics to distinguish useful subsumptions from the set of possible subsumptions.

Symbolic execution is monotonic with respect to the definition of subsumption, a result which extends to SE^*. As a result, there is no need to explore the successors of a subsumed point: they are subsumed by the successors of the subsumer. This is a crucial point to recall for proving that our algorithm preserve the set of feasible paths of the original LTS.

Theorem 1. *Let c and c' be two configurations such that $c \sqsubseteq c'$, and l a label. Then $SE\ c\ l \sqsubseteq SE\ c'\ l$.*

Proof. The proof is obtained by structural induction on l:

- it trivially follows Definition 9 when $l = \text{Skip}$,
- if $l = \text{Assume } \phi$ then, by equation (3.1):

$$\begin{aligned} States(SE\ c\ l) &= \big\{\sigma \in States(c).\ \phi(\sigma)\big\} \\ &\subseteq \big\{\sigma \in States(c').\ \phi(\sigma)\big\} \\ &= States(SE\ c'\ l) \end{aligned}$$

- if $l = \text{Assign } v\ e$ then, by equation (3.2):

$$\begin{aligned} States(SE\ c\ l) &= \big\{\sigma(v := e(\sigma)) \mid \sigma.\ \sigma \in States(c)\big\} \\ &\subseteq \big\{\sigma(v := e(\sigma)) \mid \sigma.\ \sigma \in States(c')\big\} \\ &= States(SE\ c'\ l) \end{aligned}$$

□

Detecting subsumptions during symbolic execution can turn the potentially infinite symbolic execution tree into a *finite symbolic execution graph*, whose back-edges are given by established subsumptions. However, since subsumption corresponds not to an equality but only to an inclusion of sets of program states, which entails a inclusion of sets of feasible paths starting at both configurations involved in a subsumption, adding a subsumption to the symbolic execution tree often comes at the price of introducing infeasible paths into it. A challenge is thus to accept only subsumptions that introduce a reasonable number of infeasible paths. This is addressed in Chapter 5.

3.4.2 Abstracting Configurations

In general, when the algorithm attempts at establishing a subsumption, the current configuration is not a particular case of its ancestor. The configurations at the potential subsumee and subsumer record two snapshots of the symbolic values of variables. Except for trivial loops, the symbolic values of some variables have changed between the two configurations and subsumption does not occur. In such a case, the algorithm is allowed to *abstract* the configuration at the potential subsumer in order to force the subsumption.

Abstracting a configuration means forgetting part of the information it carries. The store component of a configuration records the symbolic variables currently associated to program variables, the path predicate records constraints on these symbolic variables, expressed as a conjunction of formulae. Abstraction discards some of these formulae and there are various ways to do so: remove a set of conjuncts, compute a weaker form of the path predicate that would be implied by the current path predicates of both configurations (see for instance [30, 40]), or even updating the store. In Chapter 5, we introduce and use different ways of computing abstractions. For now, we define abstraction of a configuration in a large sense.

Definition 13. Let c be a configuration. An *abstraction* of c is a configuration c_a such that $c \sqsubseteq c_a$.

Performing an abstraction at a loop header can be seen as discovering a loop invariant for the context at this occurrence of the loop header. This procedure is a crucial point in our approach (as well as for any abstraction-based analysis technique) since the ability of detecting infeasible paths directly depends on its accuracy. Even if the strongest possible invariant could be guessed by testing all possible subsets of constraints of the path predicate at the loop header, this is not possible in practice since it is exponential. In our approach, the idea is that the invariant is a subset of the constraints in the path predicate at the loop header. Abstraction is triggered by the attempt to introduce a subsumption and we discard some of these constraints until the configuration at the loop header becomes a generalization of the configuration at the about-to-be-subsumed point. A single node might subsume any number of nodes, and can be abstracted any number of times needed. These potential subsumees exist on distinct paths, thus we discard constraints by testing path by path. This is equivalent to compute path-based invariants for each path within the loop and then intersect them at the loop header. We say that a formula ϕ is a path-based invariant along a path p if it holds in the states (or the configurations) before and after the execution of p. This prevents to compute the strongest invariants but, for performance reasons, we cannot afford an expensive method of abstraction.

Once abstraction has been performed on the subsumer, configurations located in its sub-tree must be recomputed by propagating the abstracted

configurations to their successors, in order to keep the symbolic execution tree consistent. Abstractions are propagated to successors by symbolic execution. When an abstraction is propagated to a program point that has been abstracted itself previously, one must take care to *combine both abstractions* at this program point. The way abstractions are combined depends on the method of abstraction used: methods of combination are described with methods of abstraction in Chapter 5.

By discarding parts of the path predicate, propagating an abstraction might turn unsatisfiable configurations back to satisfiable ones, making them targets for symbolic execution again.

Propagating an abstraction might rule out existing subsumptions involving successors of the subsumer: when the abstraction is propagated to an already existing subsumee, one must check that the existing subsumption still holds since the new abstraction possibly enlarges the set of states of the subsumed configuration.

Finally, once abstraction has been propagated, the new configuration of the about-to-be-subsumed node might not be subsumed by the configuration of the potential subsumer anymore, and subsumption must be checked again. When such a conflict exists, we discard the current abstraction - the attempt to subsume fails - and keep the existing subsumptions, if any.

Forcing the first possible subsumption by abstraction is not always of great profit: each abstraction discards part of the information about the symbolic values of program variables at the program point it is performed. Thus, infeasibility of some paths might not be detected anymore, and a whole new set of infeasible paths might be added to the symbolic execution graph, in comparison to the set of feasible paths one would obtain with classical symbolic execution. A crucial point to stop unfolding loops without introducing too many infeasible paths is the choice of an adequate potential subsumer — there might exist more than one other occurrence of the loop header — and an abstraction of the selected subsumer that makes that subsumption possible without discarding too many constraints of the path predicate.

Controlling Abstractions To prevent from performing unwanted abstractions, or for recording that some abstraction has been banned, predicates can be attached to configurations for loop headers. They act as safeguards against too crude abstractions: only abstract configurations that imply this additional predicate will be considered. For these abstractions to exist, such a predicate must hold for the configuration it labels, i.e it must hold for all program states represented by the configuration. This predicate could be provided by the user prior to the analysis. It could be a functional invariant of the considered loop, as is done (for example) with `assert` statements in Frama-C [37]. Such predicates are usually obtained by

some kind of counterexample guided refinement (see for example [35, 39]). In Chapter 5, we use a weakest-precondition calculus for that purpose. These additional predicates are not part of configurations and are not propagated to successors during symbolic execution.

3.5 Red-Black Graphs

At each step of the analysis, our algorithm maintains an intermediate data-structure called a *red-black graph* that we use to represent the over-approximated set of feasible paths computed so far. Informally, a red-black graph is made of two parts: *(i)* the *black part* which is simply the initial LTS, remaining unchanged during the analysis; *(ii)* the *red part*, which can be seen as a partial unfolding of the black part, decorated with subsumption links. The idea is that the red part represents the currently known set of prefixes of feasible paths. These prefixes are suffixed by sub-paths in the black part, i.e. the unknown part of the set of feasible paths. We call such paths *red-black paths*. Vertices of the red part, called *red vertices*, are indexed versions of locations of the black part: they represent the occurrences of program locations visited during the analysis.

The use of the black part can be justified as follows. Suppose an algorithm for building symbolic execution graphs that over-approximate the set of feasible paths of LTS. In order to prove that a SEG produced by this algorithm contains all the feasible paths of the original LTS, one would have to assume that this SEG has been completely built, otherwise there is a fair chance that some feasible path has not been yet followed and thus cannot be contained in the SEG. Our algorithm, like other approaches in this domain [31], uses a *refine-and-restart* mechanism to rule out too crude abstractions. Such mechanisms can cause algorithms to not terminate: once an abstraction has been refined, the algorithm restarts at the configuration where the abstraction occurred, now labeled with a safeguard condition that will prevent this abstraction to occur again, in favor of a more accurate abstraction at a latter occurrence of the loop header. In some cases, this only postpones the faulty abstraction that will happen at a later occurrence, triggering another refine-and-restart, and so on. In such cases, we force termination by bounding the length of the symbolic paths our algorithm can build, but the SEG is now only a partial unfolding of the original LTS. When this is the case, we "plug" this partial SEG and the original LTS to obtain a new graph representation containing *all feasible paths* of the original LTS. In Chapter 4 we prove that, at any moment during the construction of the red part (the SEG), the set of red-black paths, i.e. the set of paths starting in the red part and ending in the black one, contains the set of feasible paths of the original LTS. To sum up, the interest of red-black graphs is twofold: *(i)* they are a suitable mathematical object to state and prove the preservation of

feasible paths, as well as other important properties of the approach, since they always contain the known and unknown parts of the set of feasible paths and *(ii)* their red-black paths describe exactly the set of paths of the LTS obtained after plugging a partial SEG into the original LTS when the algorithm is forced to terminate.

In this section and the following, we focus describing the kernel operations performed on red-black graphs, but not on how to combine them to get an accurate LTS representation of the program under analysis (this is the topic of Chapter 5). We identify five such operations:

- symbolic execution of a black transition (see 3.6.1),
- adding a subsumption link (see 3.6.2),
- abstracting a configuration (see 3.6.3),
- marking a red vertex as unsatisfiable (see 3.6.4),
- labeling a red vertex with a safeguard condition (see 3.6.5),

and consider all other aspects of our algorithm, typically the choice of the next transition to execute, how abstractions are computed and propagated, how unsatisfiability is proven and how safeguard conditions are computed, to be heuristics and thus parameters of our approach.

We state a clear separation between the kernel transformations and the heuristics parts that combine them when trying to built an adequate red-black graph. This separation will ease the formalization presented in Chapter 4. To illustrate what we mean, consider the two following operations as examples: propagation of an abstraction in a sub-tree or cancellation of an abstraction that is found to be too crude. Both operations can actually occur in our prototype, but we do not want to formalize them since they are not needed to prove the main properties of red-black graphs. Somehow, the propagation of abstraction in the sub-tree would be unnecessary if the abstraction had been performed beforehand, when the vertex was still a leaf in the graph. Cancellation of an abstraction is always followed either by the use of a more adequate abstraction or by an unfolding at the loop header. So, we suppose that this operation is performed in the first place. Hence, the rest of this section must be read as if we were performing a smart reconstruction of the graph that our prototype had built, reconstruction freed of any unsuccessful attempt by always guessing the right transformation in the right order. We describe operators as if cancellation or propagation were never necessary, and add corresponding pre-conditions for the operators. In the following, we signal the reader when and why such restrictions are needed. For instance, in the model, abstractions only occur at leaves of the graph.

The prototype will not have the corresponding preconditions for the transformations, and the final red-black graph will be the result of a heuris-

tic search for the selection of abstractions and subsumptions, with all propagation or backtracking issues for having first try dead ends or a wrong ordering of transformations.

Ignoring subsumption links, if any, the red part is a sub-tree of the potentially infinite symbolic execution tree. It is represented as a *rooted directed graph*.

Definition 14. A *rooted directed graph* G is triple (V, r, E) where:

- V is a finite set of vertices,
- $r \in V$ is the root,
- $E \subseteq V \times V$ is a finite set of edges.

Similarly to LTS, we suppose graphs to be equipped with two applications: $src : E \to V$ and $tgt : E \to V$ respectively associating sources and targets to edges. Given a vertex v of a graph, we note $E_i(v)$ and $E_o(v)$ the sets of in-going and out-going edges of v, respectively.

When considering subsumption links, the red part can be seen as a finite symbolic execution graph, whose back edges are given by subsumption links. Red-black graphs are decorated with much information. First, they are equipped with a *subsumption relation* recording subsumption links between red vertices computed so far. A subsumption relation is a set of pairs of red vertices: an element (rv, rv') of such a relation denotes the fact that rv is subsumed by rv'. Moreover, we associate to every red vertex rv:

1. a configuration,
2. a boolean, recording information about unsatisfiability of the current configuration of rv: it is *true* if and only if the configuration has been proved unsatisfiable. In our algorithm, a constraint solver is used for that purpose. It is assumed to be correct but not complete. Since calls to the solver are expensive, it is not called upon every red vertex and its results are stored. If the solver answers with *unknown*, then the configuration is considered satisfiable, otherwise, feasible paths might be ruled out,
3. a formula over program variables: it is the safeguard condition computed so far for rv, for limiting abstractions at this program point.

Definition 15. A *red-black graph* RB is a 6-uple (B, R, S, C, M, Φ) where:

- $B = (L, l_0, \Delta, F)$ is a LTS called the *black part*,
- $R = (V, r, E)$ is a rooted directed graph called the *red part*. Its vertices, called the *red vertices*, are indexed versions of elements of L, hence $V \subseteq L \times \mathbb{N}$,

- $S \subseteq V \times V$ is a subsumption relation. Given a subsumption (rv, rv') in S, we say that rv is the *subsumee* and rv' the *subsumer*. We note $subsumees(S)$ the set of subsumees of S and $subsumers(S)$ its set of subsumers,
- C is a function from red vertices to configurations. Given a red vertex rv, we call *configuration of* rv the configuration $C(rv)$,
- M is a function from red vertices to boolean values, called *the marking*, recording partial information about unsatisfiability: given a red vertex rv, $M(rv)$ is *true* iff the configuration of rv has been proved unsatisfiable,
- Φ is a function from red vertices to formulae over program variables recording safeguard conditions used for limiting abstractions.

Initially, the red part contains no edges and a single vertex: its root, which is the first occurrence of the initial location of the black part. It is then extended using our five transformation operators.

A red vertex rv such that:

1. it is not proved to be unsatisfiable,
2. it is not subsumed by one of its ancestors,
3. there exists some black transition going out of $fst(rv)$ that has yet no red counterpart going out of rv, with $fst(p)$ (resp. $snd(p)$) standing for the first (resp. second) element of an ordered pair p,
4. it is not an occurrence of a final black location,

is considered to be linked to the black part, i.e. paths of the red part ending in such rv are implicitly suffixed by sub-paths in the black part. Such paths are formalized using the notion of *red-black paths*. The set of such red vertices constitutes what we call *the fringe* of their red-black graph. Defining these two concepts would require a number of definitions that are not needed here, such as the definitions of sub-paths and paths in graphs and graphs equipped with a subsumption relation, etc. We prefer to skip all these definitions here and to defer their presentation to Section 4.3.

Extending red prefixes with black suffixes combined with the fact that our five transformation operators never rule out feasible paths yields red-black graphs whose set of red-black paths contains every feasible path of their black part, at each step of their construction.

Example 6. The red-black graph in Figure 3.4 represent a partial unfolding of the LTS of the lock acquisition program. The configuration of red vertex 10^0 has been proved to be unsatisfiable, and 10^0 has been marked, which is shown by the $\perp$ symbol in the figure. As seen in Example 5, the configuration of 3^2 is subsumed by the configuration of its ancestor 3^0 and a subsumption

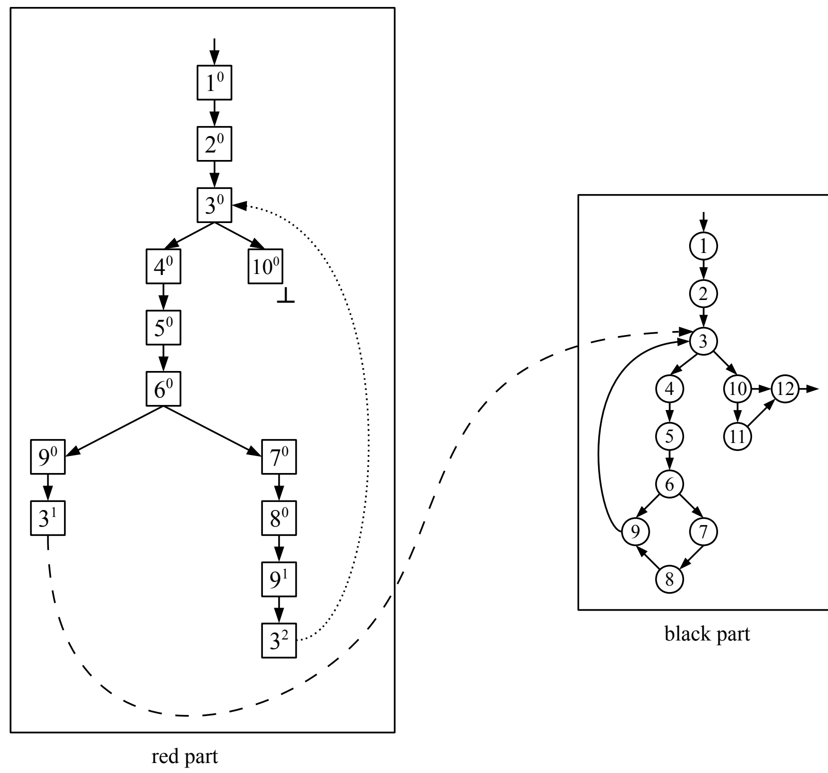


Figure 3.4: A red-black graph representing a partial unfolding of the LTS of the lock acquisition program.

link, depicted by a small-dotted edge, has been added. No transition going out from program location 3 have been executed from red vertex 3^1, which is not marked, not subsumed and not an occurrence of a final location, hence it is linked to the black part: paths going through 3^1 end in the black part. This is shown by the dashed edge. The two other leaves of the red part are either subsumed (3^2) or marked (10^0) thus they are not linked to the black part. Here, the fringe contains a single vertex, 3^1, which is the only leaf that is a neither marked nor subsumed occurrence of a non-final vertex. □

3.6 Red-Black Graphs Transformations

3.6.1 Extension by Symbolic Execution

Our first operator takes a red-black graph RB, a red vertex rv of RB and a transition δ from its black part. If a number of side-conditions are met, it adds an edge to the red part of RB as a result of symbolic execution of δ from rv. Otherwise the operator cannot apply. These conditions are:

- rv must be an occurrence of the source of δ. Otherwise, the new red part would not be an unfolding of the black part, since it would contain paths that have no counterpart in the black part,
- rv must not be already subsumed. If this is the case, then its successors are subsumed by the successors of its subsumer: there is no point in building them,
- δ can be symbolically executed only once from rv, thus, there must not exists an edge of $E_o(rv)$ whose target is an occurrence of the target of δ (we suppose LTS to contain at most one transition linking two distinct locations).

If these conditions are met, the resulting red-black graph - we call it *the extension of RB by symbolic execution of δ from rv* or, shorter, an extension of RB by symbolic execution - is obtained from RB by adding to its red part an edge linking rv to a new occurrence of the target of δ. The configuration of this new red vertex (let us call it rv') is the result of symbolic execution of the label of δ from the configuration of rv. Since we do not want to check satisfiability of configurations after each step, for performance reasons, our default action of symbolic execution w.r.t. satisfiability is to propagate current knowledge: symbolic execution propagates the marking of rv to rv'. A configuration can be marked as unsatisfiable only by two means: either by an explicit verdict from a constraint solver, or by symbolic execution from a known unsatisfiable ancestor, a useless but still correct transformation. Otherwise, the default information about satisfiability, i.e. the current path being “possibly feasible”, is propagated to successors. Hence, in all cases, a safe marking is associated with the new configuration.

Definition 16. Let $RB = \big(B, (V, r, E), S, C, M, \Phi\big)$ be a red-black graph, $rv \in V$ a red vertex of RB and δ a transition of B such that:

1. $\mathit{fst}(rv) = \mathit{src}(\delta)$,
2. $rv \notin \mathit{subsumees}(S)$,
3. $\forall e \in E_o(rv).\ \mathit{fst}\big(\mathit{tgt}(e)\big) \neq \mathit{tgt}(\delta)$.

Moreover, let rv' be the next occurrence of $\mathit{tgt}(\delta)$ in RB and e the edge (rv, rv'). The *extension of RB by symbolic execution of δ from rv*, noted $\mathit{Ext}_{SE}\ RB\ \delta\ rv$, is the red-black graph $\big(B, (V', r, E'), S, C', M', \Phi\big)$ where:

- $V' = V \cup \{rv'\}$,
- $E' = E \cup \{e\}$,
- $C' = C\big(rv' := \mathit{SE}\ C(rv)\ \mathit{label}(t)\big)$,
- $M' = M\big(rv' := M(rv)\big)$.

3.6.2 Extension by Subsumption

Our second operator adds a subsumption link to the subsumption relation. It takes a red-black graph RB and two distinct elements of its set of red vertices. We suppose that we want to add the subsumption of rv', the successor, by its ancestor rv. It requires the following conditions to be met:

- rv and rv' must be two occurrences of the same program location, since we are only interested in this kind of subsumption in practice,
- rv, the subsumer, must not be itself subsumed. Since subsuming rv' by rv implies that rv' is a descendant of rv, this would contradict the fact that rv, like any subsumed vertex, would not have been expanded.
- rv', the subsumee, must not already subsume another vertex. This would mean that rv' already has successors, again a contradiction with the fact that subsumees are supposed to have no successors. If needed, the subsumption of rv' should have been performed in the first place, without considering expending its sub-tree. The configuration that rv' is supposed to subsume can be subsumed by rv as well.
- rv' must not be already subsumed. Since a subsumed vertex is only visited once, there is no point in subsuming the same vertex more than once. If the new subsumption is more accurate than the existing one, then it must be done in the first place.
- rv' must have no out-going edges, since subsumed vertices are not supposed to have successors.
- rv and rv' must not be marked as unsatisfiable. If the configuration at rv' is unsatisfiable, it is indeed trivially subsumed by the configuration

of rv as well as by any other configuration. But since rv' does not occur along a feasible path, we are not interested in it anyway. If rv is marked, then its set of states is empty and so is the set of states for the configuration at rv' and we are back in the previous case,

- finally, for obvious reasons, the configuration of rv' must be subsumed by the configuration of rv.

Definition 17. Let $RB = \big(B, (V, r, E), S, C, M, \Phi\big)$ be a red-black graph and $rv \in V$ and $rv' \in V$ two distinct red vertices of RB such that:

1. $\mathit{fst}(rv) = \mathit{fst}(rv')$,
2. $rv \notin \mathit{subsumees}(S)$,
3. $rv' \notin \mathit{subsumers}(S)$,
4. $rv' \notin \mathit{subsumees}(S)$,
5. $E_o(rv') = \emptyset$,
6. $\neg\ M(rv)$,
7. $\neg\ M(rv')$,
8. $C(rv') \sqsubseteq C(rv)$.

The *extension of RB by addition of subsumption* (rv', rv), noted $\mathit{Ext}_{sub}\ RB\ (rv', rv)$, is the red-black graph $\big(B, (V, r, E), S', C, M, \Phi\big)$ where $S' = S \cup \{(rv', rv)\}$.

What is provided by the prototype is how we choose the target for the subsumption when there are more than one occurrence of the same program location prior in the path, or criteria for discarding a subsumption that might be established. Once a target for subsumption is selected, if an abstraction is needed for it the problem of finding which of the possibly many abstractions to select must also be tackled.

3.6.3 Extension by Abstraction

The third operator replaces the configuration of a red vertex rv by some abstraction of this configuration. This operator takes a red-black graph, a red vertex rv and a configuration that must be an abstraction of the configuration of rv. Additional constraints are as follows:

- rv must be a leaf of the red tree: as written in the introduction of this section, we do not want to formalize the propagation of an abstraction in a sub-tree, hence in this model we force abstractions to occur at leaves only.

- rv must not be marked. We are not interested in subsumptions involving an unsatisfiable configuration. The prototype never performs abstraction at a configuration known to be unsatisfiable: there is no point in applying an abstraction that could turn such a configuration back to satisfiable. In the prototype, inverting a *true* mark for a vertex can occur only as a side effect of the propagation needed by the introduction of an abstraction at some of its ancestors, during a heuristic search of a subsumption. The prototype never inverts a *true* by an abstraction at the unsatisfiable configuration itself.
- the abstraction must entail the safeguard condition associated to rv, if any. If no such condition was yet computed, we consider it to be *true* and the operator can apply.

The new red-black graph is obtained by replacing the old configuration of rv by the abstraction.

Definition 18. Let $RB = \big(B, (V, r, E), S, C, M, \Phi\big)$ be a red-black graph, $rv \in V$ a red vertex of RB and c_a a configuration such that:

1. $E_o(rv) = \emptyset$,
2. $C(rv) \sqsubseteq c_a$,
3. $\neg\ M(rv)$,
4. $c_a \models \Phi(rv)$.

where $c \models \phi$ denotes the fact that a configuration c *entails* a formula ϕ, i.e. $\forall \sigma \in States(c).\ \phi(\sigma)$.

The *extension of RB by abstraction of rv*, noted $Ext_{abs}\ RB\ c_a\ rv$, is the red-black graph $\big(B, (V, r, E), S, C', M, \Phi\big)$ where $C' = C(rv := c_a)$.

In this chapter, we are not interested in the way abstraction are computed. Of course, when presenting the prototype in Chapter 5, we will make explicit how abstractions are selected; abstractions are not guessed in advance but are triggered by the need to establish a subsumption, hence abstraction will no longer be restricted to leaves of the red part.

3.6.4 Extension by Marking

The fourth operator simply marks a red vertex of a given red-black graph if its configuration has been proved to be unsatisfiable. We use this operator to model the fact that: *(i)* the constraint solver is not called every time a vertex is added to the red part but by explicit request and *(ii)* its results are stored to avoid futur calls.

As for the other transformations, we request rv to be a leaf of the red part to save the formalization of propagating unsatisfiability notification within

a sub-graph. We also impose that rv is not subsumed since we do not want unsatisfiable subsumed configurations in our result: such a subsumption should not have ever occurred.

Definition 19. Let $RB = \big(B, (V, r, E), S, C, M, \Phi\big)$ be a red-black graph and $rv \in V$ a red vertex of RB such that:

1. its configuration $C(rv)$ has been proved unsatisfiable,
2. $E_o(rv) = \emptyset$,
3. $\neg\ M(rv)$,
4. $rv \notin subsumees(rv)$.

The *extension of RB by marking rv*, noted $Ext_m\ RB\ rv$, is the red-black graph $\big(B, (V, r, E),\ S, C, M', \Phi\big)$ where $M' = M(rv := true)$.

3.6.5 Extension by Strengthening

The last operator labels a red vertex with a safeguard condition. Safeguard conditions are not propagated to successors, but in the model limiting abstractions that can be applied at a configuration is supposed to precede any application of the abstraction operator, to avoid having to formalize the possible cancellation of an abstraction caused by a delayed introduction of a limiting predicate. Hence, strengthening is restricted to vertices that *(i)* can be the target of an abstraction, *(ii)* are not already abstracted.

Finally, this operator requires that the configuration of the about-to-be-labeled vertex entails the safeguard condition. If this was not the case, then it would be impossible to find abstractions of the current configuration that entail the safeguard condition, since abstraction consists in forgetting part of the information carried by the configuration.

If these conditions are met, this operator updates the marking of the red vertex with the given condition.

Definition 20. Let $RB = \big(B, (V, r, E), S, C, M, \Phi\big)$ be a red-black graph, $rv \in V$ a red vertex of RB and ϕ a formula over program variables such that $C(rv) \models \phi$. The *extension of RB by strengthening of rv by ϕ*, noted $Ext_{str}\ RB\ \phi\ rv$, is the red-black graph $\big(B, (V, r, E), S, C, M, \Phi'\big)$ where $\Phi' = \Phi(rv := \Phi(rv) \wedge \phi)$.

Remark: As for the other transformations, here we do not detail the way this safeguard condition is computed. In the formalization, we might even have skipped this operation since we do not explain how abstractions are computed. There is no point in explaining which abstractions have to be discarded: simply give the correct one. Without that operation, one would have

been able to prove the exact same properties we prove in the following chapter. Nonetheless, we make explicit the addition of safeguard conditions to red-black graphs to keep a closer connection between the formalization and the prototype in which this operation is performed as part of the "abstract-refine" paradigm. Safeguard conditions come from proving that some paths become unduly feasible because of a too loose abstraction.

3.6.6 The Set of Red-Black Graphs

From now on, we will only consider red-black graphs built using the five previous operators, starting from a red-black graph in an initial state. As said in Section3.5, a red-black graph is an initial state if its set of red-edges and its subsumption relation are empty, its only red vertex is its root, which is *(i)* the first occurrence of its black root, *(ii)* is not marked and *(iii)* whose safeguard condition is *true*. We talk about *well-formed* red-black graphs.

Definition 21. A red-black graph $RB = (B, R, S, C, M, \Phi)$, with $B = (L, l_i, \Delta, F)$ and $R = (V, r, E)$ is said to be *well-formed*, noted $wf\ RB$, if one of the following conditions holds:

1. $V = \{r\} \wedge r = l_i^0 \wedge E = \emptyset \wedge S = \emptyset \wedge M(r) = \mathit{false} \wedge \Phi(r) = \mathit{true}$
2. $\exists RB'\ \delta\ rv.\ wf\ RB' \wedge RB = Ext_{SE}\ RB\ \delta\ rv$
3. $\exists RB'\ rv'\ rv.\ wf\ RB' \wedge RB = Ext_{sub}\ RB\ (rv', rv)$
4. $\exists RB'\ c_a\ rv.\ wf\ RB' \wedge RB = Ext_{abs}\ RB'\ c_a\ rv$
5. $\exists RB'\ rv.\ wf\ RB' \wedge RB = Ext_m\ RB\ rv$
6. $\exists RB'\ \phi\ rv.\ wf\ RB' \wedge RB = Ext_{str}\ RB\ \phi\ rv$

From this definition, one can deduce an induction principle over well-formed red-black graphs that we use extensively to prove the key properties of our approach.

3.7 Building Red-Black Graphs: an Example

In this section, we illustrate through an example how the five previous operators can be used to build red-black graphs. Consistently with the rest of this chapter, we assume the best choice is always made and ignore how these best choices are guessed.

This example is based on the lock acquisition program seen previously. In the following figures, square vertices represent red vertices.

Initialization: The analysis starts with the red part containing no edges and only one vertex: 1^0, which is its root and the first occurrence of

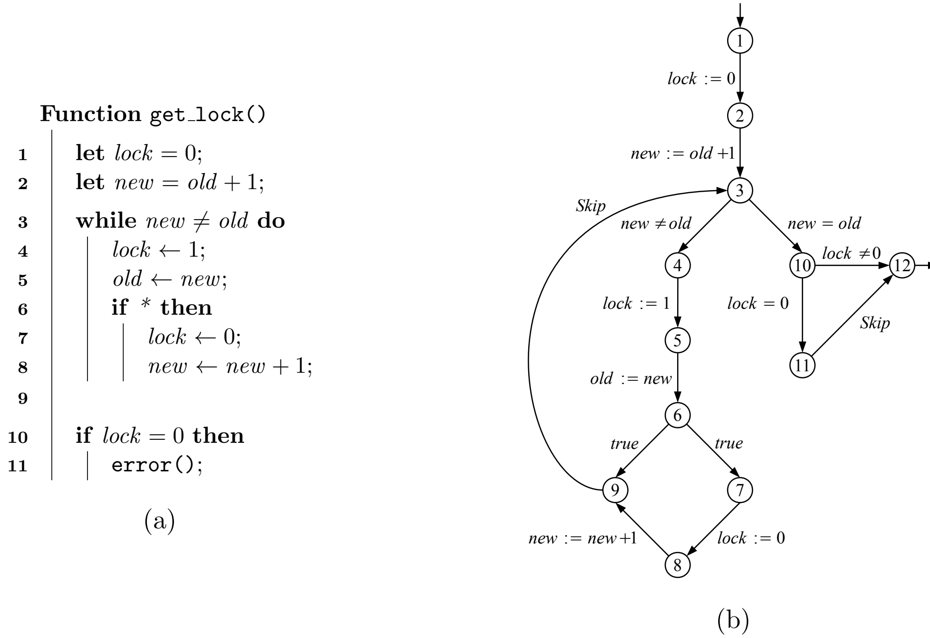


Figure 3.5: The lock acquisition program (a) and its LTS (b).

the initial location of the black part. Its configuration is $\big(\{lock \mapsto 0, new \mapsto 0, old \mapsto 0\}, true\big)$. Since 1^0 is not marked, not subsumed, not final and since none of the out-going transitions of black location 0 have been symbolically executed yet, 1^0 is linked to the black part, which is depicted by a dashed edge in Figure 3.6. At this point, the set of red-black paths is simply the set of paths of the black part.

Symbolic execution of assignments: from 1^0, classical symbolic execution is performed over the black transitions leading from 1 to 2 and from 2 to 3. This results in the addition of the red edges from 1^0 to 2^0 and from 2^0 to 3^0 (see Figure 3.7). The configuration of 2^0 and 3^0 are $\big(\{lock \mapsto 1, new \mapsto 0, old \mapsto 0\},\ lock_1 = 0\big)$ and $\big(\{lock \mapsto 1, new \mapsto 1, old \mapsto 0\},\ lock_1 = 0 \wedge new_1 = old_0 + 1\big)$, respectively.

Limiting abstractions with safeguard conditions: suppose symbolic execution continues from 3^0 by entering the loop, follows the path going along the Else-branch of the inner conditional statement and completes one iteration of the loop when it reaches the next iteration of program location 3.

The configuration obtained at this point would be $\big(\{lock \mapsto 2, new \mapsto 1, old \mapsto 1\},\ lock_1 = 0 \wedge new_1 = old_0{+}1 \wedge lock_2 = 1 \wedge old_1 = new_1\big)$

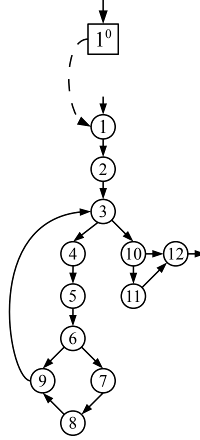


Figure 3.6: An initial red-black graph for the lock acquisition program. Its only red vertex is the first occurrence of the initial black location.

and we have seen in Example 5 that this configuration is not subsumed by the configuration at 3^0. In order to establish this subsumption, one would first have to abstract the configuration at 3^0. This would be done, for example, by weakening the path predicate at 3^0 to *true*, since both of its constraints prevent the subsumption to occur.

However, this abstraction would make the transition from 3 to 10 feasible from 3^0. Since *lock* is 0 at 3^0, this would make the error location (line 11) reachable from 3^0. In our algorithm, such an abstraction would not be allowed and the safeguard condition $new \neq old$ would be computed and associated to 3^0 to prevent the abstraction.

In Chapter 5 we explain how this abstraction would be refused and how the condition is computed. In this example, since we that best choices are always guessed, 3^0 is immediately strengthened with this condition, which is shown in Figure 3.8 (between square brackets). Future abstractions of 3^0, if any, will have to entail this safeguard condition to be considered. Since extending red-black graphs by strengthening is redundant with the assumption of best choice, we observe that we could have skip this step and simply not select this inadequate abstraction.

Symbolic execution of guards: assuming the first symbolic path enters the loop, symbolic execution is performed from 3^0 over the transition from 3 to 4. The path predicate at 4^0 is the conjunct of the path predicate at 3^0 with the adaptation of constraint $new \neq old$ to the store of the configuration of 3^0, namely $new_1 \neq old_0$. Assuming we follow first the *else* branch of the inner conditional, symbolic execu-

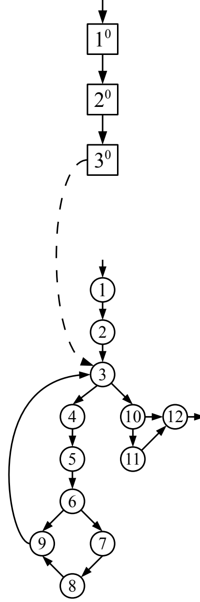


Figure 3.7: The red-black graph after the symbolic execution of the first two transitions.

tion is performed until the next occurrence of program location 3 (the loop header) is reached, hence completing one loop iteration (see Figure 3.9). The configuration for 3^1 is $(\{lock \mapsto 2, new \mapsto 1, old \mapsto 1\},\ lock_1 = 0 \ \wedge\ new_1 = old_0 + 1 \ \wedge\ lock_2 = 1 \ \wedge\ old_1 = new_1)$. As seen previously, this configuration is not subsumed by the configuration of 3^0 and the safeguard condition of 3^0 prevents any attempt of introducing an abstraction to force subsumption. As a result, symbolic execution follows and the loop is unfolded.

Marking nodes as unsatisfiable: Symbolic execution is performed from 3^1 to 4^1. The path predicate at 4^1 is unsatisfiable, as it requires *new* and *old* to be both equal and different. We *mark* node 4^1 (depicted by a $\perp$ symbol, in Figure 3.10) to represent the fact that it is known as unsatisfiable. The path leading to 4^1 being infeasible, we do not go farther along that path. Symbolic execution resumes at the last branching point, i.e. 3^1 and now follows the symbolic path that exits the loop. Since *lock* is 1 at 3^1, the first occurrence of the error location (11) is detected unsatisfiable and marked and the first occurrence of the only final location (12) is reached.

Subsumption between loop headers: symbolic execution resumes at 6^0, the next pending point. The path going along the Then-branch of the

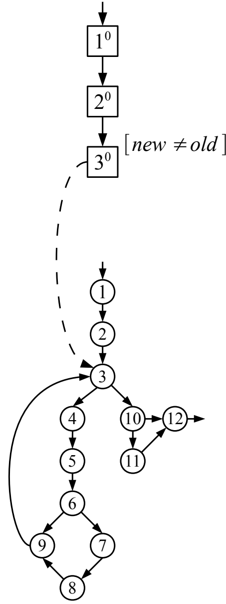


Figure 3.8: The red vertex 3^0 is strengthened by a safeguard condition that will prevent unwanted abstractions.

inner conditional statement is followed, and symbolic execution continues until the third occurrence of the loop header is reached. The configuration of 3^2 is $\big(\{lock \mapsto 3, new \mapsto 2, old \mapsto 1\},\ lock_1 = 0 \wedge new_1 = old_0 + 1 \wedge lock_2 = 1 \wedge old_1 = new_1 \wedge lock_3 = 0 \wedge new_2 = new_1 + 1\big)$ which is subsumed by the configuration at 3^0 without the need to abstract the latter. Moreover, the configuration at 3^2 entails the safeguard condition of 3^0: subsumption $(3^2, 3^0)$ is added to the subsumption relation, which is depicted by the dotted edge in Figure 3.11.

End of the analysis: the last pending point is 3^0 from which the transition from 3 to 10 is symbolically executed. The configuration at 10^0 is unsatisfiable, since its path predicate requires *new* and *old* to be both equal and different and 10^0 is marked (see Figure 3.12). Since every leaf of the red part is either subsumed, marked, or an occurrence of the final location of the black part, the red part is no longer linked to the black part: the red part is said to be complete. The set of red-black paths of this final red-black graph is simply the set of paths of the red part that do not end in marked locations. This set contains exactly the feasible paths of the original LTS: it is the set of paths that go at least one time into the loop and always take the Then-branch of the inner conditional statement, except for their last iteration.

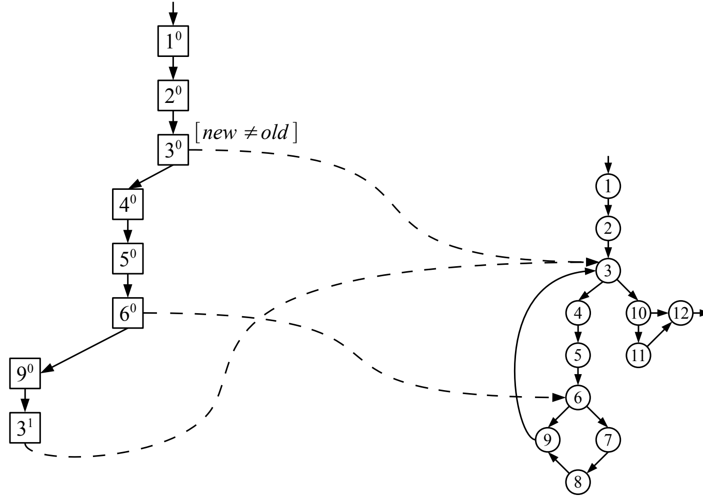


Figure 3.9: The loop has been iterated once along the first path. The subsumption of 3^1 by 3^0 is discarded by the safeguard condition of the latter.

3.8 Summary

In this chapter, we have introduced our way of modeling programs and their behaviors using LTS and symbolic execution respectively. We have presented and formalized the notion of subsumption our algorithm relies on to produce a finite symbolic execution graph when classical symbolic execution would yield an infinite symbolic execution tree.

In order to force to establish a subsumption when this would not be possible otherwise, abstraction can be performed at loop headers. The way abstraction is performed is a crucial point in our approach, since abstracting configurations might introduce approximations into the final result. We have shown how too crude abstractions can be handled by labeling loop headers met during symbolic execution by safeguard conditions.

These elements are the kernel operations of our algorithm. Moreover, this algorithm is parameterized by heuristics that are in charge of performing these operations in a way that maximizes the detection of infeasible paths.

We have formalized the data structure on which our algorithms is based by introducing the notion of red-black graph. The kernel operations mentioned above are modeled as transformation operators over red-black graphs. We claim that the nature of red-black graphs, which connect the currently known set of feasible paths (the red part) to the original LTS (the black part), combined with our five operators results in the construction of red-black graphs whose set of red-black paths contains all the feasible paths of the original model, at any step of their construction.

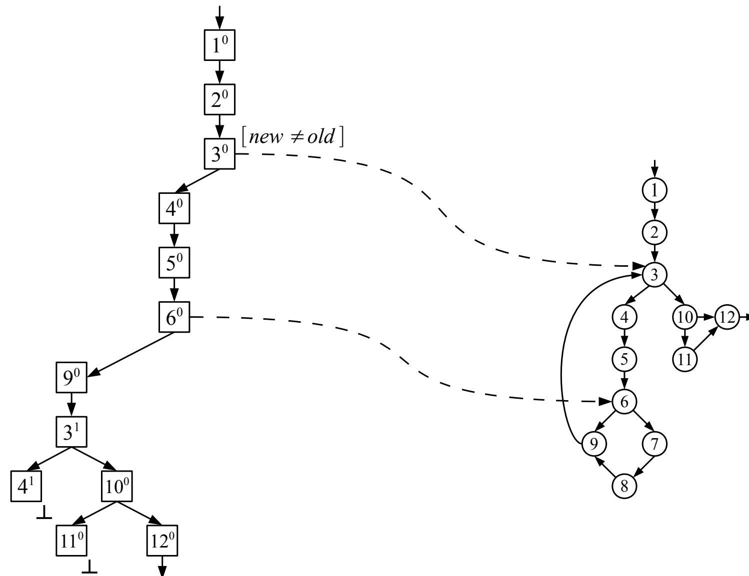


Figure 3.10: The final location has been reached and infeasible paths have been detected through marking.

What we have presented here provides the bases of a generic method for detecting and pruning infeasible paths that preserves all feasible paths of the original model. The proof of this claim is the object of the next chapter. How these operators are combined so as to maximize the detection of infeasible paths is the object of Chapter 5, where our actual algorithm is introduced.

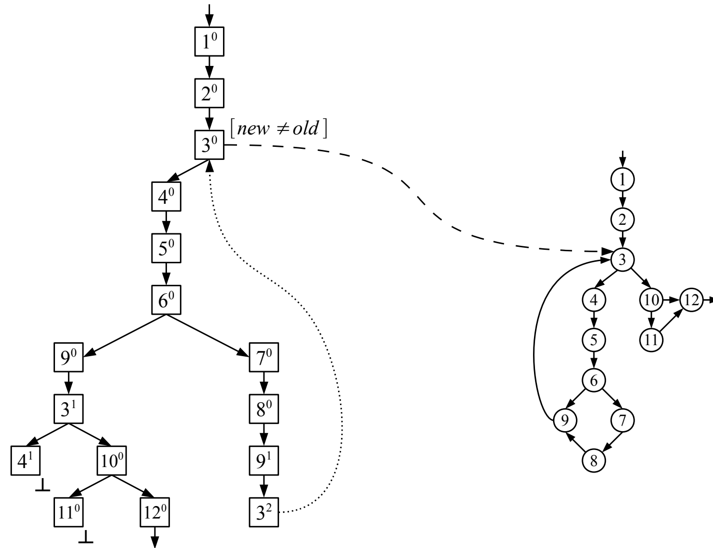


Figure 3.11: The safeguard condition of 3^0 does not prevent the subsumption of 3^2 by 3^0.

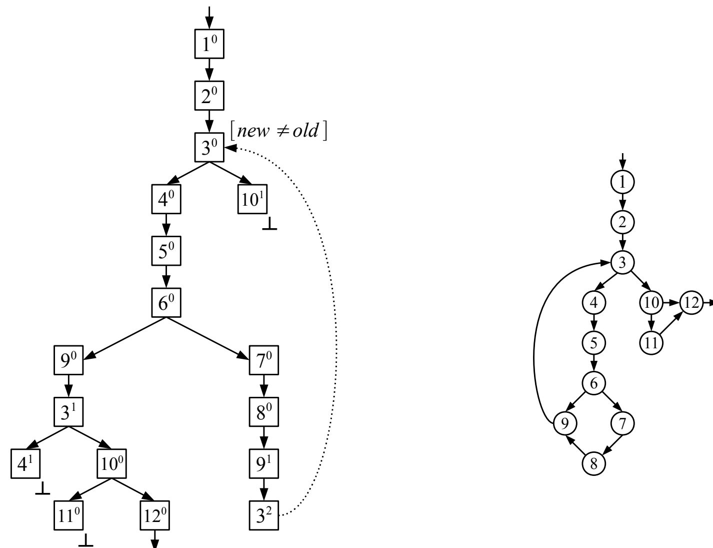


Figure 3.12: The red part is a complete unfolding of the black part, and is no longer linked to the latter.

Chapter 4

Formalization

4.1 Introduction

In this chapter, we present a formalization of our approach. The goal of this formalization is to prove that our method, besides preserving computations, produces SEGs that contain all feasible paths of the LTS representation of the program under analysis. As seen in Chapter 3, the number of details that must be considered when reasoning precisely about correctness issues of these types of graph-based algorithms is quite substantial and makes it a valuable target for machine-checked analysis. Our formalization was done using the interactive theorem proving environment Isabelle/HOL for Church's higher-order logic, a classical logic based on a simply typed λ-calculus extended by parametric polymorphism.

Proving is generally a complex task, and doing so within interactive theorem proving environments can be tedious, as it requires to consider a number of details one would take for granted when proving with pen and paper. In the following, we describe the main concepts involved in this formalization, and state the most important theorems needed to establish the proofs of the key properties of our approach. Although a real effort was made to keep proofs as short as possible — most of them are only a few lines long, it is not possible to give all the proof details in this document: some proofs are a few hundred lines long. In the following, we will give sketches that try to capture the main points in these proofs. The whole proof script, registered in the "Archive of Formal Proofs" server [4], is given in the appendix. This work was presented during the seventh conference on Interactive Theorem Proving, in August 2016 (see [3]).

Our formalization is made of twelve theories, each of them defined in its own `.thy` file, which are organized as depicted in Figure 4.1, an arrow from theory T_j to T_i denoting the fact that T_j makes use of concepts defined in T_i. When formalizing our approach, we first made a strict distinction between the symbolic execution and graph theory related aspects of the problem.

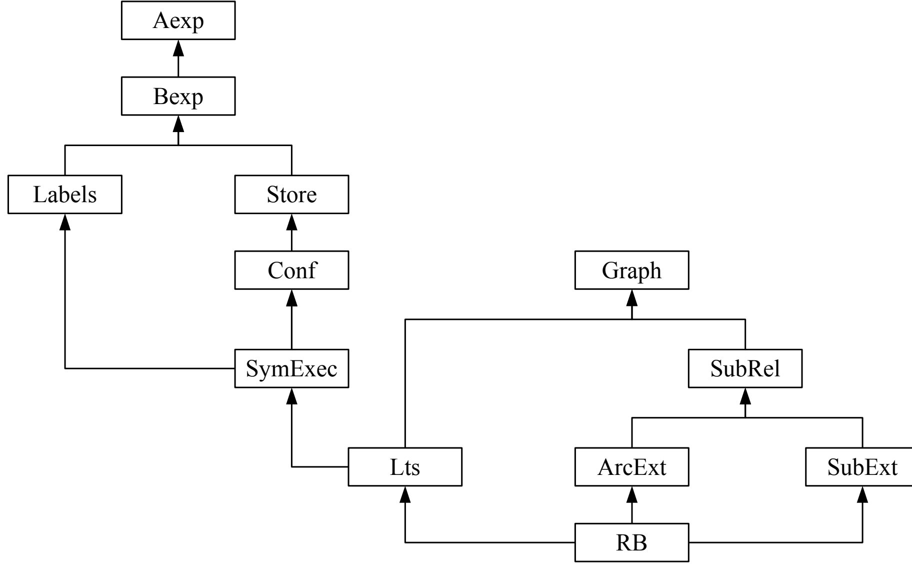


Figure 4.1: The hierarchy of theories in our formalization.

First, we model the concepts needed to define our notion of symbolic execution (`SymExec.thy`): arithmetic and boolean expressions (`Aexp.thy`, `Bexp.thy`), stores (`Store.thy`), configurations with abstraction and subsumption (`Conf.thy`). At this point, one goal is to show that our notion of symbolic execution is monotonic with respect to subsumption. This is addressed in Section 4.2.

On the other hand, we model our notions of rooted graphs, consistency of sequences of edges, sub-paths and paths in `Graph.thy`. We model LTS (`LTS.thy`) by extending rooted graphs with functions that associate labels (`Labels.thy`), our representation of program statements, to their edges. We model subsumption relations between red vertices and add them to our notion of graph, including subsumption links in alternate definitions of consistency, sub-paths and paths in `SubRel.thy`. We describe how the set of paths of a graph equipped with a subsumption relation evolves after adding a new edge or a new subsumption link in `ArcExt.thy` and `SubExt.thy`, respectively. This is the object of Section 4.3

Finally, we model red-black graphs, the five transformation operators, the concepts of fringe and red-black paths, and state and prove the key properties of our approach in `RB.thy`, to which Section 4.4 is devoted.

4.2 Symbolic Execution

In this section, we introduce our modeling of the different notions related to the symbolic execution aspects of our approach. We start with arithmetic and boolean expressions. We consider two types of expressions: expressions over program variables and expressions over symbolic variables. Modeling expressions in a syntactic manner, a usual way to do so, would make our formalization dependent on the logic of these expressions. We proceed differently: expressions are modeled by total functions, that is, by their semantics. Expressions are evaluated using program states and symbolic states, i.e. total functions associating values to program variables or symbolic variables, respectively. Our technique to represent arithmetic and boolean expressions is widely known under the term "shallow embedding", a term that goes back to [14]. This modeling of expressions requires to rethink some well-known concepts about expressions, for example freshness of a variable w.r.t. an expression, or substitution. We continue with stores — which simply map program variables to symbolic variables — and related notions: consistency of a program state and a symbolic state w.r.t. to a store, adaptation of expressions to stores (our equivalent for flat substitution). From expressions and stores, we introduce our modeling of configurations — which are made of a store and a set of boolean expressions over symbolic variables whose conjunction is the path predicate — and introduce some important notions: satisfiability and states of a configuration, subsumption, abstraction. Finally, we introduce our notion of symbolic execution steps and its main property: its monotonicity with respect to subsumption.

4.2.1 Arithmetic and Boolean Expressions

As in Chapter 3, we keep a distinction between the set of program variables *Vars* and the set of symbolic variables *SymVars*, the latter being defined as $Vars \times \mathbb{N}$. Our formalization is parameterized by two type-variables $'v$ and $'d$ representing *Vars* and D respectively.

Symbolic Variables Symbolic variables are modeled as pairs consisting of a program variable and an index:

type-synonym $'v\ symvar = {'v} \times nat$

Program and Symbolic states Program and symbolic states are modeled as total functions from a type-variable $'v$ to a type-variable $'d$, representing the type of variables and their domain, respectively.

type-synonym $('v,'d)\ state = {'v} \Rightarrow {'d}$

Unlike the previous type-synonym, $'v$ stands here for *any kind of variable*, that is program or symbolic variables. This allows to use a single type-synonym to model both program states and symbolic states.

Arithmetic and Boolean Expressions Usually, arithmetic and boolean expressions are modeled syntactically. It is very easy doing so, using Isabelle/HOL's datatypes: one would define an `Aexp` datatype and provide a constructor for each operator to include in the model. However, this has two drawbacks. First, this would require long and tedious definitions of very classical notions like terms, substitutions, valuation of an expression, etc, and that would not allow us to benefit from the existing logical machinery of Isabelle/HOL. Second, the theory would hold only for currently defined operators, requiring new definitions and lemmas each time an operator needs to be included to the model. To avoid these problems, we chose to model arithmetic and boolean expressions as total functions from states to values. In other words, expressions are modeled by their semantics.

type-synonym $('v,'d)$ *aexp* $=$ $('v,'d)$ *state* $\Rightarrow$ $'d$
type-synonym $('v,'d)$ *bexp* $=$ $('v,'d)$ *state* $\Rightarrow$ *bool*

Once again, using type-variables, a distinction is made between expressions - arithmetic or boolean - over program variables and expressions over symbolic variables. This is not a problem since we never need expressions over both types of variables.

Variables of an Expression In the rest of the formalization, we often need to reason about the variables of an expression. If we were to model expressions as structured terms, as usually done, then it would be quite straightforward to characterize the set of variables of such expressions: it would be the set of variables occurring in the expression, i.e a subset of the leaves of the term representing it. Since we model expressions as total functions from states to values, it makes no sense to say that a variable occurs in an expression. We define the set of variables of an expression as the set of variables that can influence its value [1]:

definition *vars* ::
  $('v,'d)$ *aexp* $\Rightarrow$ $'v$ *set*
where
  vars $e = \{v.\ \exists\ \sigma\ val.\ e\ (\sigma(v := val)) \neq e\ \sigma\}$

[1] These two definitions bear the same name, which is not possible in Isabelle/HOL unless they are given in separate theories. In our case, these two notions are defined in `Aexp.thy` and `Bexp.thy`, respectively, and are referred in the following theories using qualified names, unless there is no ambiguity.

definition *vars* ::
  $('v,'d)$ *bexp* $\Rightarrow$ $'v$ *set*
where
  $vars\ e = \{v.\ \exists\ \sigma\ val.\ e\ (\sigma(v := val)) \neq e\ \sigma\}$

As an example, the arithmetic expression $x-y$ is represented by the function $\lambda\ \sigma.\ \sigma\ x - \sigma\ y$. Its set of variables is $\{x, y\}$ if and only if $x \neq y$, and the empty set otherwise. Similarly, an expression like $\lambda\ \sigma.\ \sigma\ x * 0$ is considered as having no variable, as if a static evaluation of the multiplication had occurred.

Fresh Variables When reasoning about symbolic execution, one usually has to consider fresh variables. In our case, in Chapter 3, we used fresh variables for handling Assign labels using a form of *Static Single Assignment*. We start by defining freshness of variables for expressions, and will extend the concept later to configurations. A variable is said to be fresh for an expression if it is not a member of the set of variables of this expression.

abbreviation *fresh* ::
  $'v \Rightarrow ('v,'d)$ *aexp* $\Rightarrow$ *bool*
where
  $fresh\ v\ e \equiv v \notin vars\ e$

The definition of freshness for boolean expressions is defined analogously. Since the notion of freshness does not bring a whole new concept to the formalization, we use the Isabelle/HOL **abbreviation** command instead of the usual **definition** to ease the following proofs. In Isabelle/HOL, definitions are expanded by the simplifier on an explicit demand, while it is automatic for abbreviations, which makes abbreviations suitable for defining notions that merely extend existing ones.

Satisfiability Satisfiability of a boolean expression is defined as usual as the existence of a state that makes it true:

definition *sat* ::
  $('v,'d)$ *bexp* $\Rightarrow$ *bool*
where
  $sat\ e = (\exists\ \sigma.\ e\ \sigma)$

Entailment A boolean expression entails another if any state that makes it true makes the second one also true:

definition *entails* ::
  $('v,'d)$ *bexp* $\Rightarrow$ $('v,'d)$ *bexp* $\Rightarrow$ *bool* (**infix** $\models_B$ *55*)
where
  $\varphi \models_B \psi \equiv (\forall\ \sigma.\ \varphi\ \sigma \longrightarrow \psi\ \sigma)$

This definition is annotated with the syntax annotation "**infix** $\models_B$ 55": *entails* $\phi\ \psi$ and $\phi \models_B \psi$ now refer to the same expression. The **infix** keyword specifies a non-oriented infix operator: expressions of the form $a \models_B b \models_B c$ are illegal. The number 55 specifies the precedence of the construct i.e. its relative syntactic priority among all operators.

4.2.2 Stores

In Chapter 3, we defined configurations as pairs (s, π), with s a store and π the path predicate. Stores and their properties are given in `Store.thy` and configurations in `Conf.thy`: once again, this distinction is made to benefit from qualified names.

Stores are modeled as functions from program variables to natural integers representing the indexes used for modeling symbolic variables.

type-synonym $'v\ store = \ 'v \Rightarrow nat$

where $'v$ represents program variables.

Variables of a Store Given a program variable v and a store s, the symbolic variable associated to v by s is the pair $\big(v, s(v)\big)$:

definition *symvar* ::
$'v \Rightarrow\ 'v\ store \Rightarrow\ 'v\ symvar$
where
symvar v s $\equiv$ $(v, s\ v)$

The set of symbolic variables given by a store, or simply its set of symbolic variables, is its target set:

definition *symvars* ::
$'v\ store \Rightarrow\ 'v\ symvar\ set$
where
symvars s $= (\lambda\ v.\ symvar\ v\ s)$ ‘ $(UNIV::'v\ set)$

where f ‘ S stands for the image of the function f over the set S, and $UNIV::'v\ set$ for the set of all objects of type $'v$.

We skip the formal definition of freshness: a symbolic variable is said to be fresh for a store if it is not a member of its set of symbolic variables.

Consistency Consistency of a program and a symbolic state w.r.t. a store is defined as in Chapter 3:

definition *consistent* ::
$('v,'d)\ state \Rightarrow ('v\ symvar,\ 'd)\ state \Rightarrow\ 'v\ store \Rightarrow bool$
where
consistent $\sigma\ \sigma_{sym}\ s \equiv (\forall\ v.\ \sigma_{sym}\ (symvar\ v\ s) = \sigma\ v)$

Given a program state and a store, one can always build a symbolic state such that the two are consistent w.r.t. to this store, and reciprocally. This is expressed by the two following lemmas:

lemma *consistent-eq1* :
  $consistent\ \sigma\ \sigma_{sym}\ s = (\forall\ sv \in symvars\ s.\ \sigma_{sym}\ sv = \sigma\ (fst\ sv))$

lemma *consistent-eq2* :
  $consistent\ \sigma\ \sigma_{sym}\ store = (\sigma = (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ store)))$

Adaptation of Expressions to Stores As seen in Definition 9 of Chapter 3, symbolic execution of Assume and Assign labels adds expressions over symbolic variables to the path predicate of the current configuration. These expressions are obtained by substituting occurrences of program variables in the expressions carried by the labels by their symbolic counterparts given by the store. Since we represent expressions by total functions, these operations cannot be expressed as substitutions. Given an expression e and a store s, the expression we want to obtain is the function that operates the same calculus than e, but over the symbolic versions of the elements of *Vars* as given by the store s. We call this operation *the adaptation of e to s*, and define it as follows:

definition *adapt-aexp* ::
  $('v,'d)\ aexp \Rightarrow 'v\ store \Rightarrow ('v\ symvar,'d)\ aexp$
where
  $adapt\text{-}aexp\ e\ s = (\lambda\ \sigma_{sym}.\ e\ (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ s)))$

The following lemma shows that our definition of adaptation matches the usual substitution:

lemma *adapt-aexp-is-subst* :
  assumes $consistent\ \sigma\ \sigma_{sym}\ s$
  shows $(adapt\text{-}aexp\ e\ s)\ \sigma_{sym} = e\ \sigma$

The definition is analogous for boolean expressions (the operation is called *adapt-bexp*) and the same property holds.

4.2.3 Configurations, Subsumption and Abstraction

Configurations are modeled by records whose first component is a store and second component a set of boolean expressions whose conjunction is the actual path predicate.

record $('v,'d)\ conf$ =
  $store :: 'v\ store$
  $pred :: ('v\ symvar,'d)\ bexp\ set$

The fact that the *pred* component is modeled by a set is inherited from earlier versions of the formalization. In our first attempts, we modeled the process of abstracting a configuration by removing one constraint from its path predicate, which is indeed one way of abstracting configurations in practice. However, this is not the only method for doing so. For example, one could give a fresh symbolic counterpart to a given program variable. This has the effect of disconnecting this program variable from the constraints of the path predicate it is subject to, without removing these constraints from the path predicate. Since these constraints might be related to other program variables, this method of abstraction yields a better infeasible path detection power on some examples than simply removing constraints.

Since we do not want our formalization to depend on heuristics — and the way abstractions are performed can be considered as such — we finally modeled abstraction as introduced in Chapter 3, i.e. as a renaming for subsumption (see the definition *abstract* at the end of this sub-section). We plan to model the *pred* component as a boolean expression in the near future, but note that this has no incidence on the scope of this work.

Symbolic Variables of a Configuration The set of symbolic variables of a configuration is the union of the set of symbolic variables of its store with the set of symbolic variables of its path predicate:

definition *symvars* ::
  $('v,'d)\ conf \Rightarrow 'v\ symvar\ set$
where
  $symvars\ c = Store.symvars\ (store\ c) \cup Bexp.vars\ (conjunct\ (pred\ c))$

where *conjunct* E denotes the expression that evaluates to *true* given a state σ iff $\forall e \in E.\ e\ \sigma$, i.e. the conjunction of the elements of E.

As usual, a symbolic variable is fresh w.r.t. a configuration if it is not a member of its set of symbolic variables. We name *fresh-symvar* the corresponding function, skipping its formal definition here.

Satisfiability and States of a Configuration A configuration is satisfiable if its path predicate is. As in Chapter 3, the states of a configuration are the program states for which there exists a symbolic state that satisfies the path predicate and such that the two are consistent with the store:

definition *states* ::
  $('v,'d)\ conf \Rightarrow ('v,'d)\ state\ set$
where
  $states\ c = \{\sigma.\ \exists\ \sigma_{sym}.\ consistent\ \sigma\ \sigma_{sym}\ (store\ c) \wedge conjunct\ (pred\ c)\ \sigma_{sym}\}$

Subsumption We now state in Isabelle/HOL the notion of subsumption from Definition 12:

definition *subsums* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ conf \Rightarrow bool$ (**infix** $\sqsubseteq$ *55*)
where
$c_2 \sqsubseteq c_1 \equiv (states\ c_2 \subseteq states\ c_1)$

In an actual implementation, subsumption is checked by assuming the existence of a program state of the subsumee and showing, using a constraint solver, that it is also a program state of the subsumer. In order to prove that this approach is correct, we define the *semantics of a configuration* as a boolean expression describing its set of states and show that checking for subsumption is equivalent to checking entailment between semantics:

definition *sem* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ bexp$
where
$sem\ c = (\lambda\ \sigma.\ \sigma \in states\ c)$

theorem *subsum-eq-sem-entailment* :
$c_2 \sqsubseteq c_1 \longleftrightarrow sem\ c_2 \models_B sem\ c_1$

Abstraction The way configurations are abstracted is a parameter of our algorithm, since one could imagine different methods for doing so. Instead of modeling how abstractions are computed, we define a predicate expressing whether a configuration is an abstraction of another. A configuration is an abstraction of another if it subsumes it:

definition *abstract* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ conf \Rightarrow bool$
where
$abstract\ c\ c_a \equiv c \sqsubseteq c_a$

4.2.4 Symbolic Execution Steps

Labels are represented using Isabelle/HOL datatypes:

datatype $('v,'d)\ label = Skip \mid Assume\ ('v,'d)\ bexp \mid Assign\ 'v\ ('v,'d)\ aexp$

The *SE* **and** *SE-star* **predicates** We model symbolic execution steps by an inductive predicate *SE* that takes two configurations c_1 and c_2 and a label l and evaluates to *true* if and only if c_2 is a *possible result* of the symbolic execution of l from c_1.

inductive *SE* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ label \Rightarrow ('v,'d)\ conf \Rightarrow bool$
where
SE c Skip c

$| \; SE\ c\ (Assume\ e)\ (\!|\ store = store\ c,\ pred = pred\ c \cup \{adapt\text{-}bexp\ e\ (store\ c)\}\ |\!)$

$$
\begin{array}{l}
|\ fst\ sv = v \quad \Longrightarrow \\
\quad fresh\text{-}symvar\ sv\ c \Longrightarrow \\
\quad SE\ c\ (Assign\ v\ e) \\
\qquad (\!|\ store = (store\ c)(v := snd\ sv), \\
\qquad\quad pred\ = pred\ c \cup \{(\lambda\ \sigma.\ \sigma\ sv = (adapt\text{-}aexp\ e\ (store\ c))\ \sigma)\}\ |\!)
\end{array}
$$

We say that c_2 is a possible result because, in the case of an assignment, we do not want to specify here how the index of the fresh symbolic variable is chosen. In Chapter 3, we required the new symbolic variable to be fresh. If we were to specify how the index is chosen, we would have to prove, each time we have a proposition of the form $SE\ c_1$ (Assign $v\ e$) c_2, that the way the index of the new variable was chosen actually yields a fresh symbolic variable for c_1. However, *SE* can take any configuration as c_1 and there is no guarantee that any given symbolic variable is indeed fresh for c_1. Indeed, since expressions are modeled as total functions, the set of variables of the path predicate of c_1 could theoretically be *SymVars*, or contain, for a program variable v, all symbolic variables in $\{(v, n) \mid n \in \mathbb{N}\}$. By not specifying how the index is chosen but rather requiring the new symbolic variable to be fresh (see the second line in the Assign case), we will already know, when needed, that the new variable is fresh, if $SE\ c_1$ (Assign $v\ e$) c_2 holds. This kind of reasoning is called rule inversion.

Modeling *SE* as a predicate rather than a function is also more convenient once having decided to not specify how the index is chosen. If we were to model *SE* as a function, and since freshness of the new variable is a necessary condition in the case of an assignment, then this function would have to be either partial or total with a special value for the case where there exists no fresh symbolic variable for c_1. Writing *SE* as a function would require additional hypothesis and work in the forthcoming proofs. Once again, using a predicate allows for rule inversion which allows to avoid this problem: if $SE\ c_1$ (Assign $v\ e$) c_2 holds, then c_2 is the (a possible) result of *SE* as defined in Chapter 3.

Rule inversion does not solve all problems related to the existence of fresh symbolic variables. Farther in this formalization, we will sometimes have to show that given a configuration c_1 and a label l, there actually exists some configuration c_2 such that $SE\ c_1\ l\ c_2$. This clearly holds for Skip or Assume labels. If l is of the form Assign $v\ e$, the existence of c_2 depends on the existence of a fresh symbolic counterpart of v for c_1. When reasoning over red-black graphs, we will typically have to prove that an arbitrary Assign label can be symbolically executed from some red vertex of a red-black graph. This requires that there exists fresh symbolic versions of every program variable for the configurations of every red vertex of such graphs. In a sense, we want to guarantee that we will never run short of fresh symbolic variables. In Appendix A.12.4 we prove that this property

holds for all the red-black graphs we are interested in and whose following two characteristics:

- their initial configuration has a finite *pred* component, and each expression in it has a finite set of variables,
- all labels in the black part carry expressions with finite sets of variables,

provide the necessary assumptions. Symbolic execution of a label from such a configuration always yields configurations with the same property, which ensures the existence of fresh symbolic variables throughout the construction of the red part.

We extend symbolic execution to sequences of labels:

inductive *SE-star* ::
$('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *label list* $\Rightarrow$ $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
SE-star c [ ] c
| *SE* c_1 l c_2 $\Longrightarrow$ *SE-star* c_2 *ls* c_3 $\Longrightarrow$ *SE-star* c_1 (l # *ls*) c_3

where [ ] denotes the empty sequence.

Monotonicity of ***SE*** **and** ***SE-star*** The most important property of symbolic execution is that it is monotonic w.r.t. subsumption. Again, we only state the corresponding theorems here, refering to Appendixes A.6.4 and A.6.6 for their respective proofs.

theorem *SE-mono-for-sub* :
assumes *SE* c_1 l c_1'
assumes *SE* c_2 l c_2'
assumes $c_2 \sqsubseteq c_1$
shows $c_2' \sqsubseteq c_1'$

theorem *SE-star-mono-for-sub* :
assumes *SE-star* c_1 *ls* c_1'
assumes *SE-star* c_2 *ls* c_2'
assumes $c_2 \sqsubseteq c_1$
shows $c_2' \sqsubseteq c_1'$

4.3 Graphs, Labeled Transition Systems, Subsumption Relations

4.3.1 Introduction

In this section, we introduce our modeling of rooted graphs, LTS and related notions. It covers five theories in which, to ease modeling red-black graphs and their transformations, we model graphs and LTS, paths - taking into account subsumption links or not - and prove a number of facts describing

the evolution of the set of paths of a graph after the addition of an edge or a subsumption link.

We do not use any deep graph-theory in our work. In Chapter 3, we introduced a number of requirements specifying how an edge or a subsumption link could be added to the red part of a red-black graph. For example, an edge can only be added if its source is already a vertex of the red part, but not its target. Thus, lemmas describing the evolution of the paths of graphs in these cases are very specific to our approach. Moreover, we consider graphs equipped with subsumption relations and paths going through elements of these relations, which is also specific to our work. For these reasons, we did not feel the need to reuse existing developments of graph-theories in Isabelle/HOL (see for example [44] or [51]).

In the following, we consider different types of graphs (red and black) and several notions of paths (classical paths in a graph, paths going through subsumption links, feasible paths). In order to avoid duplicating definitions and lemmas, we want our model of graphs to be generic and extensible: we use a record parameterized by the type of its vertices, and later extend it with a labeling function of the edges to model LTS.

To model red-black paths as introduced in Chapter 3, we have to consider both classical paths (black paths) and paths going through subsumption links (red paths). Reasoning about classical paths is quite straightforward, but subsumption links complicate the problem. We found that it was easier to distinguish between these two notions, rather than to model subsumption links as a special case of edges.

4.3.2 Rooted Graphs

We model edges by a record that is parameterized with a type variable $'v$ representing the type of vertices.

record $'v$ *edge* =
  src :: $'v$
  tgt :: $'v$

Rooted graphs are modeled by a record containing their roots and their sets of edges.

record $'v$ *rgraph* =
  root :: $'v$
  edges :: $'v$ *edge set*

The set of vertices of a rooted graph is not a component in the record definition; we chose to deduce it from its two components as follows:

definition *vertices* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *'v set*
where
  vertices $g = \{root\ g\} \cup src$ ‘ *edges* $g \cup tgt$ ‘ *edges* g

This is justified by the fact that a record definition in Isabelle/HOL does not allow restrictions over its components: adding a "set of vertices" field to the *'v rgraph* record without the associated invariant that the source and target of each edge actually belong to this set of vertices would be of no help. This could be solved by assuming said invariant each time it is needed, but we found it simpler to deduce the set of vertices from the set of edges: the invariant is always implicitly assumed.

Isabelle/HOL provides extensible records, that is, one can define new record types by extending existing ones. Records defined in this way "inherits" the definitions and lemmas existing for the original ones. The definition of *vertices* is given for $('v,'x)$ *rgraph_scheme*, i.e. for any extension of *'v rgraph*, where $'x$ is a type variable standing for the types of additional components. Indeed, in the following, we will model LTS by extending the *'v rgraph* record with a labeling function for edges.

Consistency, Sub-Paths and Paths Sub-paths of a rooted graph are consistent sequences of some of its edges. Intuitively, a sequence of edges is consistent between two vertices if it leads from one to the other without discontinuity. The two following definitions are inspired from the Graph Library for Isabelle by Nochinsky [44].

fun *ces* ::
  $'v \Rightarrow$ *'v edge list* $\Rightarrow$ $'v \Rightarrow$ *bool*
where
  ces v_1 [ ] $v_2 = (v_1 = v_2)$
| *ces* v_1 $(e\#es)$ $v_2 = (src\ e = v_1 \wedge ces\ (tgt\ e)\ es\ v_2)$

definition *subpath* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v \Rightarrow$ *'v edge list* $\Rightarrow$ $'v \Rightarrow$ *bool*
where
  subpath $g\ v_1\ es\ v_2 \equiv ces\ v_1\ es\ v_2 \wedge v_1 \in vertices\ g \wedge set\ es \subseteq edges\ g$

Since edges elements exist independently of graphs (one can define edges without reference to any graph), consistency of a sequence of edges is not related to a given graph and is only a property of the sequence itself. On the other hand, the definition of *subpath* above takes a graph g as first parameter.

In both definitions, the two vertices v_1 and v_2 are here for several reasons. In the case of *subpath*, we will often need, in the following, to consider the starting and ending vertices of a sub-path: using them as parameters of the definition avoids us to write operators that return the endpoints of

a sequence of edges. Writing such operators would require a special treatment for the empty sequence. With our definitions there is no such problem since the starting (and thus the ending) vertex of the empty sequence are parameters of *subpath*. This is also why they are parameters of *ces*: not making them part of the signature of *ces*, would require to add the constraint $es = [\,] \rightarrow v_1 = v_2$ in the definition of *subpath*, when our goal is to have a unified definition. Also, these two additional parameters will prove particularly handy, in the following, when reasoning about sub-paths going through subsumption links.

Finally, the definition of *subpath* does not explicitly require that v_2 is a vertex of g, but implies it:

lemma *lst-of-sp-is-vert* :
 assumes *subpath g* v_1 *es* v_2
 shows $v_2 \in$ *vertices g*

Paths are sub-paths starting at the root of the given rooted graph.

abbreviation *path* ::
 $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *'v edge list* $\Rightarrow$ *'v* $\Rightarrow$ *bool*
where
 path g es v $\equiv$ *subpath g* (*root g*) *es v*

4.3.3 Labeled Transition Systems

We model LTS by extending the *'v rgraph* record. The new record is parameterized by three type variables *'vert*, *'var* and *'d* representing the types of vertices, program variables and their domain, respectively.

record (*'vert*,*'var*,*'d*) *lts* = *'vert rgraph* +
 labeling :: *'vert edge* $\Rightarrow$ (*'var*,*'d*) *label*

Objects of type (*'vert*,*'var*,*'d*) *lts* are simply rooted graphs equipped with a function associating labels to their edges: definitions and lemmas that hold for rooted graphs also hold for LTS.

Given a sequence of edges and a labeling function, the trace of this sequence is defined as follows:

abbreviation *trace* ::
 'vert edge list $\Rightarrow$ (*'vert edge* $\Rightarrow$ (*'var*,*'d*) *label*) $\Rightarrow$ (*'var*,*'d*) *label list*
where
 trace es L $\equiv$ *map L es*

We are mainly interested in feasible sub-paths and paths of LTS. Since LTS are also rooted graphs, we use the definition of *subpath* for rooted graphs:

abbreviation *feasible-subpath* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$
  $('var,'d)$ *conf* $\Rightarrow$
  'vert $\Rightarrow$
  'vert edge list $\Rightarrow$
  'vert $\Rightarrow$ *bool*
where
  feasible-subpath lts c v_1 *es* $v_2 \equiv$ *Graph.subpath lts* v_1 *es* v_2
  $\wedge$ *feasible c* (*trace es* (*labeling lts*))

A sequence of edges is feasible from a given configuration if symbolic execution of its trace yields a satisfiable configuration:

definition *feasible* ::
  $('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *label list* $\Rightarrow$ *bool*
where
  feasible c ls $\equiv$ ($\exists$ c'. *SE-star c ls* c' $\wedge$ *sat* c')

This definition is not related to LTS but to sequence of labels. As such, it is defined in `SymExec.thy` but was only introduced now since it was not needed before this point.

Again, feasible paths are defined as feasible sub-paths starting at the root of the given LTS.

4.3.4 Graphs Equipped with Subsumption Relations

Subsumptions take place between vertices of the red part of a red-black graph, that is, between occurrences of vertices of its black part.

type-synonym *'v sub-t* = $(('v \times nat) \times ('v \times nat))$

A subsumption relation is simply a set of subsumptions.

type-synonym *'v sub-rel-t* = *'v sub-t set*

Consistency, Sub-Paths and Paths We now consider sub-paths and paths going through subsumption links. This requires to re-define the notions of consistency and sub-path to take subsumption relations into account. In the following, for the sake of simplicity, we still talk about (sub-)paths of a rooted graph equipped with a subsumption relation, although we rather keep the graph and the subsumption relation as separate parameters in the following definitions.

Here, a sequence of edges is consistent between two vertices if it leads from one to the other and it is made of a number of consistent (in the sense of the previous definition) sub-sequences linked together by elements from the subsumption relation. Hence, this new definition of consistency accepts sequences that would not be consistent if it was not for subsumption links.

fun *ces* ::
  $('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow ('v \times nat) \Rightarrow {}'v$ *sub-rel-t* $\Rightarrow$ *bool*
where
  $ces\ v_1\ []\ v_2\ subs = (v_1 = v_2\ \vee (v_1,v_2) \in subs^+)$
$|\ ces\ v_1\ (e\#es)\ v_2\ subs = ((v_1 = src\ e \vee (v_1,src\ e) \in subs^+) \wedge$
$\qquad ces\ (tgt\ e)\ es\ v_2\ subs)$

In Definition 17 of Chapter 3, we imposed several restrictions on subsumption relations of red-black graphs. For example, they must contain no chain of subsumptions (this is imposed through the conditions of application of Ext_{sub}). But nothing in our above definitions of *sub_rel_t* and *ces* relates to this particular constraint. Thus, when considering a consistent sequence of edges w.r.t. a subsumption relation, one must take into account the possibility that the "gaps" might be filled by chains of subsumptions and not only individual subsumption links. Thus, the empty sequence is consistent between v_1 and v_2 w.r.t. a subsumption relation S if (v_1, v_2) is an element of the transitive closure of S. If the sequence is not empty, its "gaps" can also be filled with such chains. This extends to its endpoints: subsumption chains might allow the sequence to start (resp. end) in another vertex than the source (resp. target) of its first (resp. last) element.

The fact that we are not interested in chains of subsumptions is a requirement that is specific to the current way we build red-black graphs but is not related to the notion of consistency of sequence of edges w.r.t a subsumption relation. The additional constraint will be added only at the level of operations on red-black graphs: adding a subsumption must not introduce such chains. We think that modeling and proving is made easier by trying to be as general as possible when introducing new concepts, and then introduce specific requirements as late as possible. The idea is that subsequent lemmas are not polluted by unnecessary assumptions: as a result, their proofs must capture the exact reasons that make these propositions theorems.

The definition of sub-paths of a graph equipped with a subsumption relation is almost identical to the previous one.

definition *subpath* ::
  $(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow$
  $('v \times nat) \Rightarrow$
  $('v \times nat)$ *edge list* $\Rightarrow$
  $('v \times nat) \Rightarrow$
  $(('v \times nat) \times ('v \times nat))$ *set* $\Rightarrow$ *bool*
where
  $subpath\ g\ v_1\ es\ v_2\ subs \equiv sub\text{-}rel\text{-}of\ g\ subs$
$\qquad \wedge\ v_1 \in Graph.vertices\ g$
$\qquad \wedge\ ces\ v_1\ es\ v_2\ subs$
$\qquad \wedge\ set\ es \subseteq edges\ g$

It takes a fifth parameter: the subsumption relation *subs*, which is passed

to *ces*. It also put one additional constraint: *sub-rel-of g subs*, which states that all vertices involved by the subsumption relation *subs* are also vertices of *g*. We say that *subs* belongs to *g*. Without this restriction, that is, if *subs* was to involve other vertices than those of *g*, the definition of *subpath* would allow sub-paths to "exit" the graph at some point through subsumption links.

Using HOL "locales" The predicate *sub-rel-of* in the previous definition is defined using Isabelle/HOL's *locales.* Locales are Isabelle's mechanism to handle parametric theories. The parameters of a locale are given by a sequence of logical variable declarations together with their type-declaration (which can be polymorphic and higher-order) and a sequence of assumptions over these variables. In the body of a locale, definitions and theorems can be stated and proved depending on these logical variables, which can be seen as the "formal constants" of the theory parameter, whereas the assumptions can be seen as "formal theory" of the parameterized theory.

In contrast to type-classes, locales can be parameterized in several polymorphic types (not just one), introduce additional syntax for the logical variables and also produce, when instantiated, a prover configuration of for the instantiated theorems.

Locales are extendable, that is, one can extend an existing context into another one by adding variables or premises, for example. We do not go into the details of locales, the interested reader might refer to the Isabelle/HOL documentation.[2] In the following, we use locales as a convenient way to add complex assumptions to lemmas and to allow to refer to specific theorems when proving them.

We first define two very simple locales, *rgraph* and *sub-rel*, in which we only declare a rooted graph *g* and a subsumption relation *subs*, respectively.

locale *rgraph* =
  fixes *g* :: $('v,'x)$ *rgraph-scheme*

locale *sub-rel* =
  fixes *subs* :: $'v$ *sub-rel-t*

Finally, we define *sub-rel-of* by extending both previous locales.

locale *sub-rel-of* = *rgraph* + *sub-rel* +
  assumes *related-are-verts* : *vertices subs* $\subseteq$ *Graph.vertices g*
begin
  lemma *trancl-sub-rel-of* :
    sub-rel-of g $(subs^+)$
end

[2]The Isabelle/HOL documentation can be found at `https://isabelle.in.tum.de/`.

As an example, a theorem in this context is that all vertices in the transitive closure of *subs* belongs to *g*. This is stated (and proved) in the body of the locale.

We use this locale to prove a number of facts about (sub-)paths of a graph equipped with a subsumption relation. For instance, if a sequence of edges is consistent between two vertices w.r.t. a subsumption relation belonging to a graph, then this sequence ends in a vertex of this graph.

lemma (**in** *sub-rel-of*) *ces-imp-end-vertex* :
 assumes *ces* v_1 *es* v_2 *subs*
 assumes *set es* $\subseteq$ *edges g*
 assumes $v_1 \in$ *Graph.vertices g*
 shows $v_2 \in$ *Graph.vertices g*

The additional "(**in** *sub-rel-of*)" clause states that we assume to be in the context defined by the *sub-rel-of* locale. The third assumption is necessary to handle the case of the empty sequence. The proof is obtained by induction on *es*.

From this, it follows that the last definition of *subpath* also entails that v_2 is a vertex of *g*:

lemma *lst-of-sp-is-vert* :
 assumes *subpath g* v_1 *es* v_2 *subs*
 shows $v_2 \in$ *Graph.vertices g*

4.3.5 Extending Graphs and Subsumption Relations

Since we are interested in proving that the set of red-black paths of a well-formed red-black graph contains the feasible paths of its black part, we need a number of facts describing how the set of red-black paths evolve when one of our five operators is applied. We choose to first describe how the set of paths of the red part evolves. Only two of the five operators actually modify the set of red paths: adding a red edge and adding a subsumption. For these two operators to apply, a number of requirements must be met. For example, the new edge must have its source in the old red part but not its target, or new subsumption links must not introduce chains of subsumption in the subsumption relation. To ease writing and reading the rest of the formalization, we introduce two operators modeling the addition of an edge to a rooted graph and the addition of a subsumption link to the subsumption relation equipping a rooted graph. The requirements of these operators are exactly those of Ext_{SE} and Ext_{sub}, from Chapter 3, that are related to the addition of a new edge or a subsumption link, without considering the symbolic execution related aspects of the problem. We model these operators using predicates instead of functions: as already discussed, this is a convenient way to handle the cases where the requirements are not met.

These two extensions are defined in their own theory file (`ArcExt.thy` and `SubExt.thy`).

abbreviation *extends* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *'v edge* $\Rightarrow$ $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
  extends g e g' $\equiv$ *src e* $\in$ *Graph.vertices g*
    $\wedge$ *tgt e* $\notin$ *Graph.vertices g*
    $\wedge$ *g'* $=$ (*add-edge g e*)

abbreviation *extends* ::
  $(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow$ *'v sub-rel-t* $\Rightarrow$ *'v sub-t* $\Rightarrow$ *'v sub-rel-t* $\Rightarrow$ *bool*
where
  extends g subs sub subs' $\equiv$ *subsumee sub* $\neq$ *subsumer sub*
    $\wedge$ *fst* (*subsumee sub*) $=$ *fst*(*subsumer sub*)
    $\wedge$ *subsumee sub* $\in$ *Graph.vertices g*
    $\wedge$ *subsumee sub* $\notin$ *subsumers subs*
    $\wedge$ *subsumee sub* $\notin$ *subsumees subs*
    $\wedge$ *subsumer sub* $\in$ *Graph.vertices g*
    $\wedge$ *subsumer sub* $\notin$ *subsumees subs*
    $\wedge$ *out-edges g* (*subsumee sub*) $= \{\}$
    $\wedge$ *subs'* $=$ *subs* $\cup$ $\{$*sub*$\}$

In the first abbreviation, *add-edge g e* stands for the graph obtained from g by adding e to its edges. In the second one, *subsumee sub* and *subsumer sub* are abbreviations for *fst sub* and *snd sub*. For a subsumption relation *subs*, *subsumees subs* and *subsumers subs* represent the sets of subsumees and subsumers of *subs*, respectively.

Sub-Paths of an Extension We now establish a number of facts describing how the set of sub-paths of a rooted graph equipped with a subsumption relation evolves after adding a new edge or a new subsumption link. In order to prove the main properties of red-black graphs, it is not necessary to express the set of paths (or sub-paths) after an extension as a function of the set of paths prior to this extension. We found that it is easier to reason with individual sub-paths.

In the case of an edge, these facts are quite intuitive and easy to prove, so we do not provide details. For example, given g and g' such that *extends g e g'* holds, a sub-path of g' that does not have e in its edges is also a sub-path of g. On the other hand, if e is an edge of this sub-path, then it must be its final edge since the target of e has no outgoing edge. For the same reason, a sub-path of g' starting at the target of e must be empty.

Describing what is happening when adding a subsumption link is a bit more complicated. We proceed in two steps. First, we describe sub-paths starting at the subsumee of the new subsumption:

lemma *sp-in-extends-imp1* :
  assumes *extends g subs* (v_1,v_2) *subs'*
  assumes *subpath g* v_1 *es* v' *subs'*
  shows $es = [\,] \lor$ *subpath g* v_2 *es* v' *subs'*

Such paths are either empty or actually start at the subsumer of the new subsumption. The proof relies on the fact that v_1, as the source of a subsumption, has no out-going edge. Thus, if *es* is not empty, the source of its first edge must be a subsumer of v_1, hence v_2 since the constraints for adding the pair (v_1, v_2) to the subsumption relation imposes that v_1 had no subsumer and v_2 is not subsumed.

Let us now consider a sub-path starting at any vertex v other than the new subsumee and ending in a vertex v'. This sub-path does not go through the new subsumption link if it was already a sub-path before adding the new subsumption. If it goes through the new subsumption link, then it can be decomposed into a non-empty prefix going from v to the new subsumee that does not go through the new subsumption, and a (potentially empty) suffix going from the new subsumee to v' and that might go any number of times through the new subsumption (the symbol @ represents the concatenation operator:

lemma *sp-in-extends-imp2* :
  assumes *extends g subs* (v_1,v_2) *subs'*
  assumes *subpath g v es* v' *subs'*
  assumes $v \neq v_1$
  shows *subpath g v es* v' *subs* $\lor$ $(\exists\ es_1\ es_2.\ es = es_1\ @\ es_2$
    $\land\ es_1 \neq [\,]$
    $\land$ *subpath g v* es_1 v_1 *subs*
    $\land$ *subpath g* v_1 es_2 v' *subs'*$)$

The proof is obtained by case distinction: we first assume that the sub-path does not go through the subsumption link, then assume that it does. The first case is proved by induction on *es* and is fairly easy so we do not give the details. Proving the second one relies on a number of intermediate lemmas that are not detailed here (see Appendix A.11.3). The main point in proving that there exists a non-empty prefix from v to v_1 is v being different from v_1 to which no subsumption links lead: at least one edge is needed to reach v_1 from v.

We model the fact that a sequence of edges goes through a given subsumption with the following boolean function:

fun *uses-sub* ::
  $('v \times nat) \Rightarrow$
  $('v \times nat)$ *edge list* $\Rightarrow$
  $('v \times nat) \Rightarrow$
  $(('v \times nat) \times ('v \times nat)) \Rightarrow$ *bool*
where
  uses-sub v_1 $[\,]$ v_2 *sub* $= (v_1 \neq v_2 \land sub = (v_1,v_2))$

$|$ *uses-sub* v_1 $(e\#es)$ v_2 *sub* =
  $(v_1 \neq$ *src* $e \wedge$ *sub* $= (v_1,$*src* $e) \vee$ *uses-sub* (*tgt* e) *es* v_2 *sub*)

4.4 Red-Black Graphs and Their Properties

In this section, we first introduce our modeling of red-black graphs and their five transformation operators, giving rise to the set of *well-formed* red-black graphs. Then, we state the three key properties of our algorithm and, for each of them, give a high-level description of the formal proof presented in the appendix.

The first one describes how red vertices of a well-formed red-black graph occurring at the end of one of its red sub-paths are related. Unsurprisingly, since we consider sub-paths going through subsumption links and since abstractions are allowed, the link between such vertices is weaker than in a classical symbolic execution tree: the descendant is usually an abstraction of the descendant one would obtain in a classical symbolic execution of all paths. This property is a crucial point for proving that feasible paths are preserved: informally, this is in essence why graphs produced by our approach are over-approximations of classical SETs.

The second key property states the correctness of our approach. Our approach can be considered correct if it does not introduce in the resulting LTS paths that do not exist in the input LTS. In a sense, we prove that our approach cannot produce results that are less accurate than its inputs.

Last but not least, we state and prove that our approach preserve feasible paths of its input LTS.

4.4.1 The Type of Red-Black Graphs

Red-black graphs are modeled using a record that is parameterized by the type variables *'vert*, *'vars* and *'d* representing respectively the type of vertices of the black part, the type of program variables and their domain.

The black part is modeled by a (*'vert*,*'var*,*'d*) *lts*, the red one by a rooted graph whose vertices are indexed instances of *'vert*. The function associating configurations to red vertices is modeled by *confs*, the subsumption relation by *subs*, the markings by *marked* and the function labeling red vertices with safeguard conditions by *strengthenings*.

We add another component to this record: *init-conf* that stores the configuration that is initially associated to the root of the red part. The configuration of the red root might be abstracted during the analysis: the sole purpose of this component is to keep track of the initial configuration to allow, in the following, to state and prove that the set of feasible paths starting at the root in this initial configuration is a subset of the red-black paths of the given red-black graph.

record $('vert,'var,'d)$ *pre-RedBlack* =
  red :: $('vert \times nat)$ *rgraph*
  black :: $('vert,'var,'d)$ *lts*
  subs :: $'vert$ *sub-rel-t*
  init-conf :: $('var,'d)$ *conf*
  confs :: $('vert \times nat) \Rightarrow ('var,'d)$ *conf*
  marked :: $('vert \times nat) \Rightarrow$ *bool*
  strengthenings :: $('vert \times nat) \Rightarrow ('var,'d)$ *bexp*

Since the components of instances of this record can take any value, these instances represent a much larger set than the set of red-black graphs that are built using only our five operators. We talk about *pre-red-black graphs.*

4.4.2 Well-Formed Red-Black Graphs

As said in Chapter 3, we are only interested in red-black graphs obtained using our five transformation operators from a red-black graph whose red part is initially empty. In this section, we describe our modeling of the five operators over red-black graphs introduced in Section 3.6.

Extension by Symbolic Execution

We model Ext_{SE} (as well as the other operators) by a predicate that takes as inputs a red-black graph *rb*, a red edge *re*, a configuration *c* and a second red-black graph rb' and evaluates to *true* if the following conditions are met:

- *re* is a "red version" of a black transition of *rb*. We note *ui-edge re* the black edge (*fst*(*src re*), *fst*(*tgt re*)),
- the red part of rb' has been obtained by an extension (by addition of an edge) of the red part of *rb*,
- the source of the new red edge is not already subsumed in *rb*,
- the configuration *c* is indeed a possible result of the symbolic execution of the label of the black version of *re* (as given by the labeling function of the black part) from the configuration associated to the source of the new red edge,
- the target of *re* is "correctly" marked in rb'.

abbreviation *se-extends* ::
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
  $('vert \times nat)$ *edge* $\Rightarrow$
  $('var,'d)$ *conf* $\Rightarrow$
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
  se-extends rb re c rb' $\equiv$
    ui-edge re $\in$ *edges* (*black rb*)
  $\wedge$ *ArcExt.extends* (*red rb*) *re* (*red rb'*)

$$
\begin{array}{l}
\wedge\ \textit{src re} \notin \textit{subsumees}\ (\textit{subs rb}) \\
\wedge\ \textit{SE}\ \ (\textit{confs rb}\ (\textit{src re}))\ (\textit{labeling}\ (\textit{black rb})\ (\textit{ui-edge re}))\ c \\
\wedge\ rb' = (\!|\ \begin{array}{ll}
\textit{red} & = \textit{red rb}', \\
\textit{black} & = \textit{black rb}, \\
\textit{subs} & = \textit{subs rb}, \\
\textit{init-conf} & = \textit{init-conf rb}, \\
\textit{confs} & = (\textit{confs rb})\ (\textit{tgt re} := c), \\
\textit{marked} & = (\textit{marked rb})(\textit{tgt re} := \textit{marked rb}\ (\textit{src re})), \\
\textit{strengthenings} & = \textit{strengthenings rb}\ |\!)
\end{array}
\end{array}
$$

Once again, the fact that we use a predicate instead of a function to model this operator is a convenient way to handle the cases where the requirements are not met. Moreover, its inputs differ from those of the operator Ext_{SE} introduced in Chapter 3: this slight modification is done in order to facilitate proofs.

Extension by Subsumption

Operator Ext_{sub} is modeled by the predicate *subsum-extends* that takes as inputs a subsumption and two red-black graphs and evaluates to *true* if the second has been obtained by extending the first one by addition of the given subsumption. Its requirements are exactly the same as those of Ext_{sub}.

abbreviation *subsum-extends* ::
  ('vert,'var,'d) pre-RedBlack ⇒
  'vert sub-t ⇒
  ('vert,'var,'d) pre-RedBlack ⇒ *bool*
where

$$
\begin{array}{l}
\quad \textit{subsum-extends rb sub rb}' \equiv \\
\qquad \textit{SubExt.extends}\ (\textit{red rb})\ (\textit{subs rb})\ \textit{sub}\ (\textit{subs rb}') \\
\quad \wedge\ \neg\ \textit{marked rb}\ (\textit{subsumer sub}) \\
\quad \wedge\ \neg\ \textit{marked rb}\ (\textit{subsumee sub}) \\
\quad \wedge\ \textit{confs rb}\ (\textit{subsumee sub}) \sqsubseteq \textit{confs rb}\ (\textit{subsumer sub}) \\
\quad \wedge\ rb' = (\!|\ \begin{array}{ll}
\textit{red} & = \textit{red rb}, \\
\textit{black} & = \textit{black rb}, \\
\textit{subs} & = \textit{insert sub}\ (\textit{subs rb}), \\
\textit{init-conf} & = \textit{init-conf rb}, \\
\textit{confs} & = \textit{confs rb}, \\
\textit{marked} & = \textit{marked rb}, \\
\textit{strengthenings} & = \textit{strengthenings rb}\ |\!)
\end{array}
\end{array}
$$

Extension by Abstraction

Operator Ext_{abs} is modeled by *abstract-extends* that takes as inputs a red vertex *rv*, a configuration c_a and two red-black graphs and evaluates to *true* if the second has been obtained by updating the *confs* component of the first one at the leaf *rv* with c_a, which must subsume the previous configuration at *rv* and entail its current safeguard condition.

abbreviation *abstract-extends* ::
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
$('vert \times nat)$ $\Rightarrow$
$('var,'d)$ *conf* $\Rightarrow$
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where

$$
\begin{aligned}
& \textit{abstract-extends}\ rb\ rv\ c_a\ rb' \equiv \\
& \quad rv \in \textit{red-vertices}\ rb \\
& \wedge \neg\ \textit{marked}\ rb\ rv \\
& \wedge\ \textit{out-edges}\ (\textit{red}\ rb)\ rv = \{\} \\
& \wedge\ rv \notin \textit{subsumees}\ (\textit{subs}\ rb) \\
& \wedge\ \textit{abstract}\ (\textit{confs}\ rb\ rv)\ c_a \\
& \wedge\ c_a \models_c (\textit{strengthenings}\ rb\ rv) \\
& \wedge\ \textit{finite}\ (\textit{pred}\ c_a) \\
& \wedge\ (\forall\ e \in \textit{pred}\ c_a.\ \textit{finite}\ (\textit{vars}\ e)) \\
& \wedge\ rb' = \ (\!|\ \textit{red} = \textit{red}\ rb, \\
& \qquad \textit{black} = \textit{black}\ rb, \\
& \qquad \textit{subs} = \textit{subs}\ rb, \\
& \qquad \textit{init-conf} = \textit{init-conf}\ rb, \\
& \qquad \textit{confs} = (\textit{confs}\ rb)(rv := c_a), \\
& \qquad \textit{marked} = \textit{marked}\ rb, \\
& \qquad \textit{strengthenings} = \textit{strengthenings}\ rb\ |\!)
\end{aligned}
$$

This abbreviation includes two particular requirements: *finite* $(\textit{pred}\ c_a)$ and $(\forall\ e \in \textit{pred}\ c_a.\ \textit{finite}\ (\textit{vars}\ e)$. The first one states that the path predicate of the new configuration c_a contains only a finite number of expressions; the second that each of these expressions must have a finite set of variables. Recall that, at some point in this formalization, we will have to prove that we are always be able to find fresh symbolic variables for configurations that are built during the analysis. The two operators over red-graphs that directly modify configurations are adding a red edge and abstracting a configuration: those two operators must be applied in a way that guarantees the existence of fresh symbolic variables in the rest of the analysis. To handle the case of symbolic execution, we will assume, when needed (see Section 4.4.3 for example), that our analysis starts with an LTS whose labels only carry expressions with finite set of program variables and from an initial configuration whose path predicate contains a finite number of expressions each of which contains only finite number of symbolic variables. A part of our formalization that is not detailed in this chapter (see Appendix A.6.7) is devoted to prove that, in these conditions, our notion of symbolic execution yields configurations with finite path predicates and whose expressions contain a finite number of variables, that is, for which there exists fresh symbolic variables. Since we do not specify how abstractions are computed, we have no choice than to require abstractions to meet these two requirements.

Extension by Marking

Operator Ext_m is modeled by the following predicate that takes as inputs a red vertex *rv* and two red-black graphs and evaluates to *true* if the second has been obtained by marking the leaf *rv* in the first one.

abbreviation *mark-extends* ::
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
  $('vert \times nat)$ $\Rightarrow$
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
  mark-extends rb rv rb' $\equiv$
    $rv \in$ *red-vertices rb*
  $\wedge$ *out-edges* (*red rb*) *rv* $= \{\}$
  $\wedge$ $rv \notin$ *subsumees* (*subs rb*)
  $\wedge$ $rv \notin$ *subsumers* (*subs rb*)
  $\wedge$ $\neg$ *sat* (*confs rb rv*)
  $\wedge$ *rb'* $=$ (| *red* $=$ *red rb*,
      black $=$ *black rb*,
      subs $=$ *subs rb*,
      init-conf $=$ *init-conf rb*,
      confs $=$ *confs rb*,
      marked $=$ (λ *rv'*. *if rv'* $=$ *rv then True else marked rb rv'*),
      strengthenings $=$ *strengthenings rb* |)

Extension by Strengthening

Operator Ext_{str} is modeled by a predicate that takes a red vertex *rv* and a (safeguard) condition *e* and two red-black graphs and evaluates to *true* if the second has been obtained by adding the new condition to the *strengthenings* component of the first with *e* at leaf *rv*.

abbreviation *strengthen-extends* ::
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
  $('vert \times nat)$ $\Rightarrow$
  $('var,'d)$ *bexp* $\Rightarrow$
  $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
  strengthen-extends rb rv e rb' $\equiv$
    $rv \in$ *red-vertices rb*
  $\wedge$ $rv \notin$ *subsumees* (*subs rb*)
  $\wedge$ *confs rb rv* $\models_c$ *e*
  $\wedge$ *rb'* $=$
    (| *red* $=$ *red rb*,
      black $=$ *black rb*,
      subs $=$ *subs rb*,
      init-conf $=$ *init-conf rb*,
      confs $=$ *confs rb*,
      marked $=$ *marked rb*,

$$strengthenings = \\ \quad (strengthenings\ rb)(rv := (\lambda\ \sigma.\ (strengthenings\ rb\ rv)\ \sigma \wedge e\ \sigma))\|)$$

The Set of Well-Formed Red-Black Graphs

The set of well-formed red-black graphs is exactly the set of red-black graphs built using our five operators, starting from a red-black graph whose red part is empty. We do not directly model this set, but rather define an inductive predicate that precisely describes it: well-formed red-black graphs are exactly those that satisfy the following predicate.

inductive *RedBlack* ::
 $('vert,'var,'d)\ pre\text{-}RedBlack \Rightarrow bool$
where
 base :
 $fst\ (root\ (red\ rb)) = init\ (black\ rb) \implies$
 $edges\ (red\ rb) = \{\} \implies$
 $subs\ rb = \{\} \implies$
 $(confs\ rb)\ (root\ (red\ rb)) = init\text{-}conf\ rb \implies$
 $marked\ rb = (\lambda\ rv.\ False) \implies$
 $strengthenings\ rb = (\lambda\ rv.\ (\lambda\ \sigma.\ True)) \implies RedBlack\ rb$

| *se-step* :
 $RedBlack\ rb \implies se\text{-}extends\ rb\ re\ p'\ rb' \implies RedBlack\ rb'$

| *mark-step* :
 $RedBlack\ rb \implies mark\text{-}extends\ rb\ rv\ rb' \implies RedBlack\ rb'$

| *subsum-step* :
 $RedBlack\ rb \implies subsum\text{-}extends\ rb\ sub\ rb' \implies RedBlack\ rb'$

| *abstract-step* :
 $RedBlack\ rb \implies abstract\text{-}extends\ rb\ rv\ c_a\ rb' \implies RedBlack\ rb'$

| *strengthen-step* :
 $RedBlack\ rb \implies strengthen\text{-}extends\ rb\ rv\ e\ rb' \implies RedBlack\ rb'$

When given such a predicate, Isabelle/HOL automatically states and prove a number of lemmas that can then be reused when proving facts involving red-black graphs. The induction principle over well-formed red-black graphs mentioned in Section 3.6.6 is one of these: it will be particularly helpful for proving facts that are presented in the following.

4.4.3 Relation Between Red Vertices

The main key properties of well-formed red-black graphs state how vertices at the end of a red sub-path are related. In a classical symbolic execution

tree, the configuration at the end of a sub-path would be the result of symbolic execution of the trace of this sub-path from the configuration at its beginning. This property is too strong for the red part of a red-black graph since we also consider sub-paths that go through subsumption links, and since configurations along those might have been abstracted.

In red-black graphs, the configuration at the end of a sub-path usually only subsumes the configuration one would obtain by symbolic execution, which is expressed by the following theorem.

theorem (**in** *finite-RedBlack*) *SE-rel* :
  assumes *RedBlack rb*
  assumes *subpath* (*red rb*) rv_1 *res* rv_2 (*subs rb*)
  assumes *SE-star* (*confs rb* rv_1) (*trace* (*ui-es res*) (*labeling* (*black rb*))) *c*
  shows $c \sqsubseteq$ (*confs rb* rv_2)

This proposition holds only in the context of the *finite-RedBlack* locale:

locale *finite-RedBlack* = *pre-RedBlack* +
  assumes *fn-init-pred* : *finite* (*pred* (*init-conf prb*))
  assumes *fn-init-pred-symvars* : $\forall$ $e \in$ *pred* (*init-conf prb*). *finite* (*Bexp.vars e*)
  assumes *fn-lts* : *finite-lts* (*black prb*)

In this locale, we introduce the assumption mentioned in Section 4.2.4 that allow to prove that we never run short of fresh symbolic variables when building the red part of *rb*.

In the previous theorem, the term (*ui-es res*) represents the sequence of black edges obtained from the red edge sequence *res* by discarding the indexes in the sources and targets of its elements.

The proof of this theorem is obtained by induction over well-formed red-black graphs. The initial case is trivial. The only possible red sub-path is the empty one, going from and to the red root: the proof is immediate thanks to subsumption being reflexive.

In the case of an extension by symbolic execution, one observes that rv_1 and rv_2 can either be "old" red vertices or the target of the new edge. If they are both old vertices, then *res* does not go through the new edge: it is a sub-path in the old red part and the induction hypothesis applies. If rv_1 is the target of the new edge, then *res* is empty and $rv_2 = rv_1$ since the latter has no out-going edges: the property is trivially true since subsumption is reflexive. In the last case, the new edge occurs exactly once in *res*, at its end. The proof is obtained by applying the induction hypothesis on the part of *res* that precedes its last step, and then showing that the property propagates to the target of the new edge thanks to the monotonicity of *SE*.

The difficulty comes with the extension by subsumption. First, we suppose that rv_1 is the new subsumee. Thus, by theorem *sp-in-extends-imp1*

introduced in Section 4.3.5, *res* is either empty, or a sub-path in the extension that starts at the subsumer. In the first case, either $rv_2 = rv_1$, and we conclude by reflexivity of subsumption, or rv_2 is the new subsumer: we conclude by transitivity of subsumption. If *res* is not empty, then we reason by backward induction on *res*. The initial case states that *res* is empty: we proceed as previously. The inductive case states that *res* is of the form $res' \cdot re$. The (internal) induction hypothesis applies on res' and we conclude using transitivity of subsumption.

When rv_1 is not the new subsumee then, thanks to theorem *sp-in-extends-imp2*, we have that *res* can be decomposed into a prefix that is a sub-path in the old red graph and a suffix that is a sub-path in the new red part. The induction hypothesis applies on the prefix, but we have to perform another backward induction on the suffix. As in the previous backward induction, we conclude thanks to *SE-star* being transitive.

The proofs for the tree other operators are easy, since these operators do not modify configurations of the red part (except for the abstraction, but then transitivity of subsumption immediately proves the property).

4.4.4 Preservation of Behaviours

Our goal being to build better over-approximations of the set of feasible paths of the program under analysis than its CFG, our approach can be considered correct if its results do not contain paths that do not exist in the original graph. The fact that our approach is indeed correct is fairly intuitive: red-edges are simply indexed versions of the black ones and are added to the red part in a consistent manner w.r.t. the black part, as well as subsumption links. Stating and proving this correctness property first requires to define the set of red-black paths, which in turn is based on the notion of *"fringe"*.

Given a red-black graph, the fringe is the set of its red vertices from which the set of feasible paths of the black part could be further refined. Vertices of the fringe are those that are neither subsumed nor marked and from which there exists a black transition that has no red counterpart yet.

definition *fringe* ::
  $('vert,\ 'var,\ 'd)\ \textit{pre-RedBlack} \Rightarrow ('vert \times nat)\ set$
where
  $\textit{fringe}\ rb \equiv \{rv \in \textit{red-vertices}\ rb.\ rv \notin \textit{subsumees}\ (\textit{subs}\ rb)\ \wedge$
  $\neg\ \textit{marked}\ rb\ rv \ \wedge$
  $\textit{ui-edge}\ \text{`}\ \textit{out-edges}\ (\textit{red}\ rb)\ rv \subset \textit{out-edges}\ (\textit{black}\ rb)\ (\textit{fst}\ rv)\}$

The fringe of a red-black graph in an initial state consists only of its red root. Adding an edge to the red part might enlarge or reduce the fringe, depending on the fact that there still exist or not transitions to be executed from the source and the target of the new edge. Adding a subsumption link or marking a red vertex always reduce the fringe, since it does not introduce

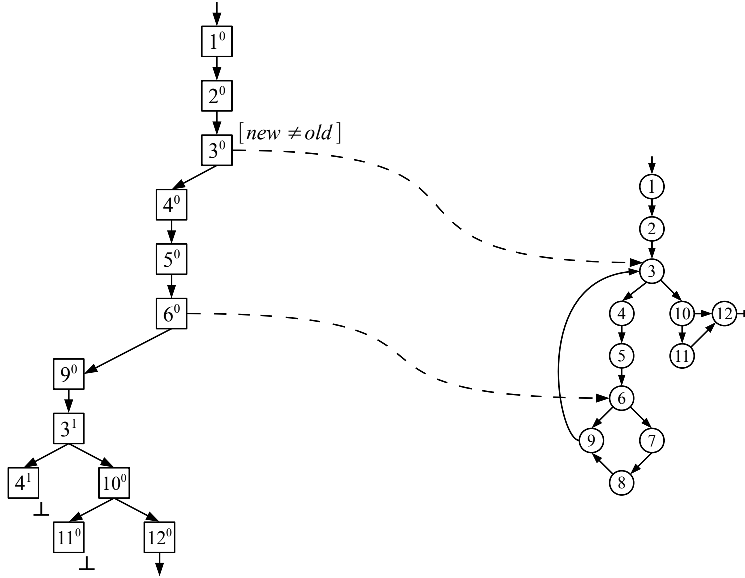


Figure 4.2: Red-black graph of the lock acquisition program with a partial red part.

successors. Seven lemmas, that are not detailed here (see Appendix A.12.7), are needed to fully describe how the fringe evolves in each case.

Example 7. The red part of the red-black graph in Figure 4.2 depicts a partial unfolding of the lock acquisition program introduced in Chapter 3. The fringe of this red-black graph is made of vertices 3^0 and 6^0, which is depicted by the dashed edges linking them to the black part. □

After having stated the effect on the fringe of each operator, we can now define the sets of red-black sub-paths and paths. Given a red-black graph *rb* and one of its red vertices *rv*, the set of red-black sub-paths of *rb* starting at *rv* is defined as the union of the two following sets[3]:

- the set of sequences of black edges obtained by unindexing the sources and targets of the elements of red sub-paths of *rb* starting in *rv* and ending in a non-marked red vertex,
- the set of sequences of black edges that have a (potentially empty) prefix that is represented in the red part by a red sub-path starting in *rv* and ending in a vertex of the fringe. Also, we require (and motivate why below) this prefix to be as long as possible.

[3]The terms "red-black sub-paths" and "red-black paths" are a bit inaccurate since these sets are expressed only in terms of black sub-paths and paths. Defining them as sets of black paths makes it easier to express and prove the theorems to come.

definition *RedBlack-subpaths* ::
  $('vert, 'var, 'd)$ *pre-RedBlack* $\Rightarrow$ $('vert \times nat) \Rightarrow$ *'vert edge list set*
where
  RedBlack-subpaths-from rb rv $\equiv$

$$
\begin{aligned}
& \textit{ui-es} \text{ ` } \{res.\ \exists\ rv'.\ \textit{subpath}\ (\textit{red rb})\ rv\ res\ rv'\ (\textit{subs rb}) \wedge \neg\ \textit{marked rb rv'}\} \\
& \cup \{\textit{ui-es}\ res_1\ @\ bes_2 \\
& \quad |\ res_1\ bes_2.\ \exists\ rv_1.\ rv_1 \in \textit{fringe rb} \\
& \qquad \wedge\ \textit{subpath}\ (\textit{red rb})\ rv\ res_1\ rv_1\ (\textit{subs rb}) \\
& \qquad \wedge\ \neg\ (\exists\ res_{21}\ bes_{22}.\ bes_2 = \textit{ui-es}\ res_{21}\ @\ bes_{22} \\
& \qquad\qquad \wedge\ res_{21} \neq [] \\
& \qquad\qquad \wedge\ \textit{subpath-from}\ (\textit{red rb})\ rv_1\ res_{21}\ (\textit{subs rb})) \\
& \qquad \wedge\ \textit{Graph.subpath-from}\ (\textit{black rb})\ (\textit{fst}\ rv_1)\ bes_2\}
\end{aligned}
$$

Once again, red-black paths are defined as red-black sub-paths starting at the red root of the given red-black graph.

In this definition, *subpath-from g v es subs* and *Graph.subpath-from g v es* stand for $\exists\ v'.\ \textit{subpath}\ g\ v\ es\ v'\ \textit{subs}$ and $\exists\ v'.\ \textit{Graph.subpath}\ g\ v\ es\ v'$, respectively.

This complex definition ensures that what we call the set of red-black paths is not trivially the set of paths of the black part, which would defeat the point of this formalization. The fact that we exclude sub-paths that end in marked vertices is quite natural since we want the set of red-black paths to be as close as possible to the set of feasible paths, and we know that sub-paths ending in marked vertices are infeasible. If we were not to require the red prefix to be as long as possible (which is expressed in the previous definition by the third conjunct in the second set, which states that the black suffix must have no non-empty red prefix) then the set of red-black sub-paths would contain black paths that are in fact known to be infeasible, since vertices of the fringe are not necessarily leaves of the red part and might have marked descendants.

For example, without this additional requirement, paths of the black part depicted in Figure 4.2 going through the following sequences of vertices:

- $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 9 \cdot 3 \cdot 4$, and
- $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 9 \cdot 3 \cdot 10 \cdot 11$,

would be red-black paths, since their common prefix is represented by a red sub-path that ends in 6^0 which is in the fringe. However, these two sequences are known to be infeasible, since they have red equivalents that end in marked vertices (4^1 and 11^0). By asking the black suffixes to have no red prefix — and thus the red prefixes to be as long as possible — the two previous paths are ruled out from the set of red-black paths, since $6 \cdot 9 \cdot 3$ and $6 \cdot 9 \cdot 3 \cdot 10$ are also represented in the red part of Figure 4.2. This does not rule out black paths ending in 9^0, 3^1, 10^0 and 12^0: they are represented by red-paths ending in non-marked red vertices (i.e. they are elements of the first set of the definition of red-black paths).

The first set in *RedBlack-subpaths* contains sub-paths of the red part starting in *rv* that were not detected infeasible yet (that is, that do not go through marked vertices). The second set contains those red sub-paths that start in *rv* and that are allowed to continue in the black part, if the red one has not been completely built yet. The red prefix might be empty, since *rv* might be itself in the fringe, and the red-black path might go "directly" into the black part.

In order to prove that our approach is correct, we first show that red sub-paths starting in a given red vertex of a well-formed red-black graph are also black sub-paths starting at the corresponding black vertex, modulo unindexing the sources and targets of their elements.

theorem *red-sp-imp-black-sp* :
  assumes *RedBlack rb*
  assumes *subpath* (*red rb*) rv_1 *res* rv_2 (*subs rb*)
  shows *Graph.subpath* (*black rb*) (*fst* rv_1) (*ui-es res*) (*fst* rv_2)

In this theorem, *ui-es res* stands for *map ui-edge es.* This intermediary theorem is a direct consequence of the definitions of *se-extends* and *subsum-extends* and is proved from the observation that for a well-formed red-black graph *rb*:

- if a sequence of red edges *res* is consistent w.r.t. the subsumption relation of *rb*, then the sequence of black edges obtained by unindexing the sources and targets of the elements of *res* is also consistent without considering the subsumption relation of *rb* (i.e. in the sense of the first definition of consistency). This is due to the fact that the subsumptions in *rb* only involve occurrences of the same black vertices,
- *fst*(rv_1) is a black vertex of *rb* and that unindexing the sources and targets of the elements of *res* yields a sequence of black edges of *rb*.

The fact that our approach is correct is expressed by the following theorem:

lemma *RedBlack-paths-are-black-paths* :
  assumes *RedBlack rb*
  shows *RedBlack-paths rb* $\subseteq$ *Graph.paths* (*black rb*)

where *Graph.paths g* is defined by comprehension from the definition of *Graph.path.* Its relies on the fact that the red root of a well-formed red-black graph is an indexed version of its black root and that, thanks to theorem *red-sp-imp-black-sp*:

- a red-black path obtained from a red path is a path in the black part,
- a red-black path made of a red prefix and a black suffix is also a path in the black part, since its red prefix can be turned into a black path leading to the same black vertex the black suffix starts at: their concatenation is a path in the black part.

4.4.5 Preservation of Feasible Paths

We are now ready to prove that our approach preserves feasible paths of the original LTS. We first show that, given a red vertex of a well-formed red-black graph *rb*, the feasible sub-paths starting at the corresponding black vertex from the configuration at *rv* are contained in the set of red-black sub-paths starting at *rv*.

theorem (**in** *finite-RedBlack*) *feasible-subpaths-preserved* :
  assumes *RedBlack rb*
  assumes $rv \in$ *red-vertices rb*
  shows *feasible-subpaths-from* (*black rb*) (*confs rb rv*) (*fst rv*)
    $\subseteq$ *RedBlack-subpaths-from rb rv*

This is the most important theorem in our formalization. Its proof is fairly difficult, and represents almost a third of the whole formalization. We proceed by induction over well-formed red-black graphs.

When *rb* is an initial state, its set of red-black paths is simply the set of black paths, since its red part is empty.

Suppose that rb' is a possible extension of *rb*, and *bes* a black feasible sub-path of rb' starting at the black vertex $fst(rv)$ of rb' (thus it is also a black feasible sub-path of *rb*), where *rv* is a red vertex of rb'. The idea is to show that *bes* is a red-black sub-path of rb'. Once again, the two difficult cases are those of the addition of a red edge and the addition of a subsumption link.

In the case of the addition of a red edge *re* at the red vertex *rv*, one has to handle a number of sub-cases. First, the black feasible sub-path might start at the target of the new edge. If there exist no black edges going out of $fst\big(tgt(re)\big)$, then *bes* must be empty. Thus, it is represented in the red part of rb' by the empty red sequence from $tgt(re)$, which cannot be marked since it just has been added to the red part. If there exist such black edges, then $tgt(re)$ is in the fringe of rb', and the empty red sequence is a suitable prefix for making *bes* a red-black subpath. If *rv* is an old red vertex, then once again one must deal with a number of sub-cases. The black feasible sub-path might not go through the new edge. Then depending, if it ends or not at the source of *re*, one must show that either it is also a red-black sub-path of rb', either that it can be extended by the new edge *re* to form red-black sub-path of rb'. If it goes through the new edge, then once again we must show that the previous red suffix can be extended by the new edge to form a red-black sub-path in rb'.

The case of the addition of a subsumption is the most complicated one. First, by induction hypothesis, we have that *bes* is a red-black sub-path of *rb*. If *bes* is entirely represented in the red part of *rb*, then it is also represented in the red part of rb' and is thus a red-black sub-path of the latter. If *bes* is made of a red prefix and a black suffix, we must consider if

this red prefix ends or not in the new subsumee. If this is the case then, since the subsumee is not in the new fringe anymore, we must find a suitable red prefix and suitable black suffix for *bes* to show that it is indeed a red-black sub-path of rb'. This is done by backward induction on the black suffix of *bes* and, again, gives rise to a number of sub-cases that we do not detail here but the idea is to show that, no matter how many times the new red prefix must go through the new subsumption link, it is always possible to reach the fringe of rb' from the new subsumer and to find a suitable red prefix and a suitable black suffix. If the red prefix ends in any other vertex than the new subsumee, then the proof is quite straightforward: this last red vertex is still in the fringe and not marked, and the known red prefix and black suffix are still suitable.

The three other cases are also quite straightforward. In the case of an extension by marking, we proceed by showing that the black feasible sub-path could not go through the newly marked vertex, otherwise it would not have been feasible in the first place. Thus it is still either entirely represented by a red sub-path ending in a non-marked vertex, or its red prefix and its black suffix are still suitable for making it a red-black sub-path of rb'.

The case of the abstraction is proved by observing that it can only enlarge the set of red-black sub-paths, thus feasible sub-paths can not be ruled out.

Strengthening a red vertex with a safeguard condition does not modify the fringe directly, nor the set of red sub-paths: it only restricts future uses of abstraction. Thus, this extension does not modify the set of red-black sub-paths by itself, and the property is trivially true, using the induction hypothesis.

From the previous theorem, we obtain that feasible black paths starting from the configuration at the red root are contained in the set of red-black paths of *rb*.

theorem (**in** *finite-RedBlack*)
  assumes *RedBlack rb*
  shows *feasible-paths* (*black rb*) (*confs rb* (*root* (*red rb*)))
  $\subseteq$ *RedBlack-paths rb*

A last point is to handle: the configuration at the root of the red part might not be the initial configuration in which the analysis started anymore, since this initial configuration might have been abstracted several times. In a well-formed red-black graph, the initial configuration is subsumed by the configuration of the red root.

lemma *init-subsumed* :
  assumes *RedBlack rb*
  shows *init-conf rb* $\sqsubseteq$ *confs rb* (*root* (*red rb*))

This is not a problem, since subsumption entails the inclusion of sets of feasible paths (modulo a number of requirements about the finiteness of sets of symbolic variables, which are met in the case of well-formed red-black graphs using the *finite-RedBlack* locale):

lemma *subsums-imp-feasible* :
assumes *finite-labels ls*
assumes *finite* (*pred* c_1)
assumes *finite* (*pred* c_2)
assumes $\forall$ $e \in$ *pred* c_1. *finite* (*Bexp.vars e*)
assumes $\forall$ $e \in$ *pred* c_2. *finite* (*Bexp.vars e*)
assumes $c_2 \sqsubseteq c_1$
assumes *feasible* c_2 *ls*
shows *feasible* c_1 *ls*

Since feasible paths considering the initial configuration are also feasible from the actual configuration at the red root, they are also contained in the set of red-black paths:

lemma (**in** *finite-RedBlack*)
assumes *RedBlack rb*
shows *feasible-paths* (*black rb*) (*init-conf rb*) $\subseteq$ *RedBlack-paths rb*

4.5 Summary

In this chapter, we introduced the main concepts and theorems established in our formalization to prove the key properties of our approach, namely that it is correct and that feasible paths of the original LTS are preserved. After introducing the symbolic execution related aspects of the problem: modeling arithmetic and boolean expressions, stores, configurations and subsumption, we formally described the symbolic execution steps themselves. The most important property of symbolic execution is its monotonicity w.r.t. subsumption. We then introduced our modeling of graphs, LTS, subsumption relation and a number of notions and theorems to describe the evolution of the set of paths of a graph equipped with a subsumption relation after an edge or a subsumption link has been added. Finally, we presented our modeling of red-black graphs and their five transformation operators, defined the notion of fringe and red-black paths and proved the key properties of our approach.

In this formalization, we abstracted a number of features of our algorithm: we do not consider the traversal strategy of the original LTS, how abstractions and safeguard conditions are computed, how candidates for subsumption are chosen, for example. This has the advantage that this formalization does not only hold for the algorithm we propose, but for a whole family of algorithms based on symbolic execution, detection of subsumptions and abstraction. Moreover, instead of reasoning on a rather complex

algorithm, involving heuristics, backtracking, refine-and-restart and propagation mechanisms, we consider the various kernel operations only, which makes proving properties of our approach far easier. Last but not least, the concept of red-black graphs proved to be particularly suitable to reason over the set of feasible paths, a rather elusive notion. In particular, the set of red-black paths describe exactly the set of paths of the results provided by our algorithm.

This formalization constitutes a solid mathematical foundation for the algorithm we propose and describe in the next chapter. Since the heuristics parts of our approach have not been modeled in this work, it is not possible to extract from this formalization, in its current state, an implementation of our approach using the code generators provided by Isabelle/HOL. Extending the formalization in this sense is definitely a heavy task and would require a substantial amount of additional work.

Proving these key properties gives a qualitative rather than quantitative feedback: our approach produces a LTS that cannot be less accurate than its input and that contains all its feasible paths. However, its infeasible path detection power obviously depends on the heuristics used to traverse the original LTS, to choose candidates for subsumption, to compute (or limit!) abstractions, etc. In the two following chapters, we fully detail these heuristics and our algorithm, and present and interpret results obtained with our prototype, whose implementation closely follows the formalization introduced in this chapter.

Chapter 5

Algorithm

5.1 Introduction

In Chapters 3 and 4, we have presented the different operations that the algorithm can perform on red-black graphs, without describing the heuristics aspects of our approach and how these operations are combined in order to detect as many infeasible paths as possible. This is the object of the present chapter where we give the full details of our algorithm.

We first recall in Section 5.2 some definitions that were introduced in Chapter 3 and that will serve us again in this presentation. We also introduce here the parameters of our approach that are dedicated to drive the various heuristics our algorithm relies on.

Our algorithm is fully described in Section 5.3. This algorithm maintains an intermediate red-black graph during all the analysis and corresponds to a DFS traversal of the black part (the input LTS) with the intent to build its red part, as described in the previous chapter. During the analysis, one of the three following actions can be performed at the current red vertex: *(i)* trigger a counterexample guided refinement phase to rule out a too crude abstraction performed previously; *(ii)* establish a subsumption of this red vertex by another and *(iii)* build the successors of this red vertex using symbolic execution. These three actions correspond to combinations of the five basic operators over red-black graphs that were introduced in the two previous chapters. Once the analysis is over, the algorithm returns a LTS whose set of paths is exactly the set of red-black paths of the intermediate red-black graph.

Section 5.3 is organized as follows. We first describe in 5.3.1 the main procedure that is responsible for traversing the input LTS and to choose which of the three actions to apply at each visited red vertex. Each action is detailed in its own sub-section. Then, we introduce in 5.3.5 a look-ahead mechanism that greatly improves the infeasible path detection power of our approach. We conclude the description of the algorithm in 5.3.6, by describ-

ing how the resulting LTS is built from the intermediate red-black graph.

Then, in Section 5.4 we illustrate how our algorithm behaves on the example of a merging sort program. This program is mainly made of three unbounded loops that iterates over one or both of its input arrays. Its LTS contains a great majority of infeasible paths, whose infeasibility is usually due to complex dependencies between iterations of the three loops. The goal of this section is to illustrate how the main features of the algorithm are combined and interact: we describe the different LTS computed for this merging sort example with various combinations of values for the input parameters that guide the heuristics during the analysis. In Chapter 6 we will report on these experiences, giving actual numbers for the ratio of infeasible paths removed and the size of resulting LTS, and we will present and discuss more experiments on different kinds of examples. Our algorithm and these experiments were presented in an internal report [1], and during the second international conference on Software Quality, Reliability and Security, in August 2016 (see [2]).

5.2 Data Structures and Inputs

5.2.1 Data Structures

Labeled Transition Systems In Chapter 3, we have defined LTS as quadruples of the form (L, l_i, Δ, F) with:

- L a set of program locations,
- $l_i \in L$ the initial location,
- $\Delta \subseteq L \times \mathit{Labels} \times L$ the transition relation, *Labels* being the set of labels,
- $F \subseteq L$ the set of final locations.

We now add to LTS a fifth attribute $LH \subseteq L$, the set of loop headers. LTS are also equipped with the following applications:

- $\mathit{src} : \Delta \to L$, $\mathit{label} : \Delta \to \mathit{Labels}$ and $\mathit{tgt} : \Delta \to L$ which, given a transition, returns its source, its label and its target, respectively,
- Δ_i and Δ_o which, given a location of L, returns its set of in-going and out-going transitions, respectively.

Red-Black Graphs During the analysis, the algorithm maintains an intermediate red-black graph RB, which is turned into the resulting LTS at the end of the analysis. Red-black graphs were defined as sextuples (B, R, S, C, M, Φ) where:

- $B = (L, l_i, \Delta, F)$ is a LTS,

- $R = (V, r, E)$ is a rooted graph, with $V \subseteq B \times \mathbb{N}$, $r \in V$ and $E \subseteq V \times V$; it is equipped with the applications *src*, *tgt*, E_i and E_o which have the same purposes than their LTS-equivalents,
- $S \subseteq V \times V$ is a subsumption relation,
- C is a function associating configurations to elements of V, i.e. red vertices; configurations are defined as pairs (s, π) where the store s is a function from program variables to indexes and π a boolean expression over symbolic variables,
- M is the marking function, associating boolean values to red vertices,
- Φ is the function associating safeguard conditions to vertices of R.

This definition must now be slightly extended to accommodate the heuristic aspects of our system. We no longer suppose, as in Chapter 4, that our final LTS is produced by knowing in advance the right sequence of operations to apply, but via some heuristic search for subsumptions and abstractions, controlled by a counterexample guided refinement mechanism. The implementation of the latter, described in the following pages, requires that we keep track of: *(i)* the initial configuration of the analysis, needed for the *refine and restart* mechanism; *(ii)* configurations that are associated to vertices of the red part R during the analysis. In this chapter, we consider that red-black graphs are of the form $(c_i, B, R, S, C, M, \Phi)$, where c_i is the initial configuration and C is a function that associates stacks of configurations, rather than a single configuration, to vertices of R. Given a red vertex *rv*, we call *configuration of rv* the element on top of the stack $C\ rv$. Also, we now consider that B is equipped with the fifth attribute LH.

In our implementation, S, C, M and Φ are implemented by compact tables to reduce the complexity of basic access and modification operations on such objects. In this presentation, we consider S to be a set and C, M and Φ to be total functions, for the sake of simplicity.

5.2.2 Inputs and Parameters

Our algorithm takes as inputs a LTS $\mathcal{S}$ and a user-provided formula *pre* over the program variables of $\mathcal{S}$, i.e. a precondition of the program under test. In the following, we suppose we have access to some constraint solver that is supposed to be correct, although not complete. The precondition, as well as expressions occurring in the labels of $\mathcal{S}$, are supposed to belong to the logic supported by this solver.

Our algorithm relies on a number of heuristics that are driven by additional parameters:

- a boolean flag, *restart*, indicates if the counterexample guided refinement mechanism mentioned earlier is enabled or not,

- a boolean flag, *dp*, indicates, when trying to subsume a vertex *rv*, which potential subsumers are considered: if *dp* is set to *false*, we only consider subsumers that occur along the path that leads to *rv*; if set to *true*, we also consider subsumers that occur on different symbolic paths. As we will see in the following chapter, allowing or not subsumption links between different symbolic paths can have a major impact on the accuracy of the resulting LTS,
- a non-negative integer *mrl* gives the maximal length of red paths the algorithm is allowed to follow; when set to 0, the length of red paths is unbounded,
- a non-negative integer *la*, *the look-ahead depth*, whose purpose will be described later.

A last parameter indicates how abstractions are performed and combined during their propagation. We do not name this last parameter: we simply suppose in the following that the corresponding functions are called.

We consider these parameters to be global variables to avoid passing them as additional parameters throughout all functions calls in our pseudocode.

Finally, a global data structure, *rvs*, of red vertices to visit is maintained during the analysis. We consider *rvs* to be a stack built according to a DFS traversal of the input LTS.

In the following we introduce our algorithm and detail its main features. We describe how symbolic execution steps are performed and how subsumptions are detected. This requires introducing abstraction calculus and propagation. Then, we introduce the counterexample guided refinement, and finally we show how to improve the infeasible path detection power of the algorithm by restricting potential subsumers during subsumption search.

5.3 Building the Red-Black Graph

5.3.1 Principles

The main procedure, `build`, depicted in Algorithm 2, is a loop that iterates until there is no more red vertices to visit in *rvs*, in which case the analysis is complete and the resulting LTS is built and returned. It takes the LTS $\mathcal{S}$ and the precondition *pre* as inputs.

In the pseudocode of `build` (and in the other functions below), the `/* CS */` marks explicitly denote where we rely on a constraint solver, be it for checking that a configuration is satisfiable, subsumed by another, or entails a given safeguard condition.

Algorithm 2: `build`

```
    input : a LTS S, a condition over program variables pre
    output: a LTS
 1  let RB = init_RB(S, pre);
 2  push(r, rvs);
 3  while ¬ empty(rvs) do
 4  |   let rv = pop(rvs);
 5  |   if ¬ M rv then
 6  |   |   let p = path_to(rv, R);
 7  |   |   if fst(rv) ∈ F then
 8  |   |   |   if restart then
 9  |   |   |   |   if infeasible(p, c_i, B) then                          /* CS */
10  |   |   |   |   |   let rv' = faulty_abs(p, B, C);                     /* CS */
11  |   |   |   |   |   Φ ← Φ(rv' := Φ rv' ∧ safeguard_cond(p, rv', B));
12  |   |   |   |   |   restore_conf(rv', C);
13  |   |   |   |   |   destroy_sub_graph(rv', R, S, C, M, Φ);
14  |   |   |   |   |   push(rv', rvs);
15  |   |   else if ¬ detect_sub(rv, RB) ∧ (mrl = 0 ∨ length(p) < mrl) then
16  |   |   |   build_se_succs(rv, B, R, C, M);                            /* CS */
17  return build_LTS(RB);
```

First, the red-black graph RB to be built is initialized as follows by the call to `init_RB` (line 1):

- c_i is the initial configuration: its store is defined over the set of program variables occurring in $\mathcal{S}$ and associates 0 to each of these variables; its path predicate is the adaptation of *pre* to this store,
- B is $\mathcal{S}$, the input LTS,
- R has no edges and a unique vertex: its root $(l_i, 0)$, the first occurrence of the initial location of B,
- S is empty,
- C associates the one-element stack containing c_i to the red root $(l_i, 0)$
- M associates *false* to $(l_i, 0)$ and Φ associates *true* to $(l_i, 0)$,[1]

and the red root $(l_i, 0)$ is pushed on top of the stack of vertices to visit *rvs* (line 2).

The analysis starts at line 3 by processing the elements of *rvs* until it is empty. For each non-marked red vertex *rv* (see line 5) processed during the DFS traversal of the black part (i.e. the input LTS), three distinct actions can take place: *(i)* if *rv* is an occurrence of a final location of the black part (line 7) and if the counterexample guided refinement is enabled through the boolean flag *restart* (line 8), we check if satisfiability of the configuration of

[1] All red vertices will have these default values for M and Φ at their creation.

rv is only due to some previous abstraction performed along the shortest path leading to *rv* (this path is built at line 6) and in that case we trigger a *refine-and-restart* phase (line 8 to 14); *(ii)* if *rv* is not an occurrence of a final location, we try to *establish a subsumption link* for *rv* in the call to `detect_sub` at line 15 (this can only succeed at occurrences of loop headers, see the pseudocode of `detect_sub` in Algorithm 3). If the subsumption is established, nothing remains to be done for *rv*; *(iii)* if none of the previous cases apply, we *build the successors* of *rv* using symbolic execution, provided that the length limit *mrl* has not been reached along the current path.

We detail below the three possible actions in the following order. First we describe how successors of a red vertex are built by symbolic execution. Then, we show how subsumptions are detected, how abstractions are computed, propagated and combined during their propagation. We continue with the counterexample guided refinement mechanism mentioned earlier, and then show how the infeasible path detection power of the overall approach can simply be improved. Finally, we show how the final LTS is computed from the red-black graph *RB* once the analysis is over.

5.3.2 Symbolic Execution Steps

This is the nominal (and simplest) action when the two special cases (final location reached and subsumption established) do not apply and the bound *mrl* has not been reached yet. Procedure `build_se_succs` (line 16) is devoted to this task. It extends the partial unfolding R of B by adding one red edge for each transition in $\Delta_o\big(fst(rv)\big)$. We only need to detail here how R, C, and M are modified at this step. For every transition $\delta \in \Delta_o\big(fst(rv)\big)$:

- the red vertex $\big(tgt(\delta), i\big)$ is added to V, where i is a fresh index for location l with respect to its previous occurrences in V; we use a global set that associates the lastly used index for any given red vertex for that purpose,
- the edge $\big(rv, (tgt(\delta), i)\big)$ is added to E,
- the configuration *SE* `top`($C\ rv$) $label(\delta)$ is pushed on the stack $C\ \big(tgt(\delta), i\big)$. If $label(\delta)$ is of the form Assign $v\ e$, the index of the new symbolic variable is simply the successor of the last index in use for v. If $label(\delta)$ is of the form Assume ϕ, with ϕ being *false*[2], then $\big(tgt(\delta), i\big)$ is marked in M. If ϕ is neither *false* nor *true*, then a constraint solver is called to check the satisfiability of the new configuration: $\big(tgt(\delta), i\big)$ is marked in M only if the solver proves it to be unsatisfiable. In the case of an *unknown* answer from the solver, the target is not marked and we consider the corresponding path to be feasible.

[2]This would occur only in case of a loop or conditional with a *true* condition in the original CFG.

- Φ is extended to $\big(tgt(\delta), i\big)$: if the latter has been marked, its safeguard condition is *false*, and *true* otherwise.

Newly built red vertices are pushed onto *rvs* for them to be processed in the next iterations of the loop.

5.3.3 Detecting Subsumptions

Let *rv* be a non-marked occurrence of a loop-header of B. Before building its successors, the algorithm attempts to establish a subsumption link between *rv* and a previously met occurrence of $fst(rv)$ that is neither marked nor subsumed. This is done by the call to procedure `detect_sub`, whose pseudo code is given in Algorithm 3. For the sake of simplicity, we consider for the moment that the execution of statements at lines 20, 21 and 22 is not conditioned by the check at line 19: we will come back to it and explain its purpose later.

Algorithm 3: `detect_sub`

input : a red vertex *rv*, a red-black graph *RB*
output: a boolean value

```
17 if fst(rv) ∈ LH then
18     foreach rv' ∈ sub_candidates(rv, R) do
19         if compare_fp_sets(rv, rv', B, C, la) then            /* CS */
20             if top(C rv) ⊑ top(C rv') then                    /* CS */
21                 S ← S ∪ {(rv, rv')};
22                 return true;
23             if abstract(rv', rv, Φ rv') = Some(a) then         /* CS */
24                 if propagable(a, rv', (rv, rv'), RB) then
25                     propagate(a, rv', R, S, C, M);
26                     S ← S ∪ {(rv, rv')};
27                     return true;
28 return false;
```

Natural subsumption The first check (line 17) verifies that *rv* is indeed an occurrence of a loop header. If this is the case then, for each previous occurrence rv' of $fst(rv)$ that is *(i)* not marked; *(ii)* not already subsumed and *(iii)* that is not in the stack of vertices to visit anymore (we call such red vertices candidates, see line 18), the algorithm checks that the configuration of *rv* can be directly subsumed by the configuration of rv' (line 20). If the check succeeds, the new subsumption link (rv, rv') is added to S (line 21) and `detect_sub` returns *true*, which prevents building the successors of *rv* in `build`. In the case of the *unknown* answer from the solver, the current

subsumption is refused: accepting it might cause some feasible paths not to be included in the result.

When searching for potential subsumers, the procedure only considers those occurrences of $\mathit{fst}(\mathit{rv})$ that are not in the stack of vertices to visit because those are more likely to be already labeled by a safeguard condition other than *true*, as seen in Section 3.4.2, avoiding to consider uninteresting abstractions. On the opposite, occurrences of $\mathit{fst}(\mathit{rv})$ that are in the stack *rvs* are still labeled by the default *true* safeguard condition: since they are in the stack, their descendants have not been built yet.

We do not give the pseudocode of the procedure `sub_candidates` called at line 18: it simply searches among red vertices of RB those previously met occurrences of $\mathit{fst}(\mathit{rv})$ that are neither marked nor subsumed. We will discuss in the following sections how the scope of candidates can be restricted in order to improve the infeasible path detection power of the approach. For example, we will see that considering subsumption links that only involve two red vertices occurring on the same path consistently yields more accurate LTS, although larger. In some cases however, relaxing this constraint allows additional sharing in the resulting LTS without loss of accuracy w.r.t. infeasible paths.

Forced subsumption If the subsumption check fails, the algorithm first attempts to build a suitable abstraction of the configuration of rv' (line 23), i.e. a configuration that subsumes the configuration of rv and that entails $\Phi\ rv'$, the safeguard condition of rv'. If such an abstraction cannot be found, another previous occurrence of $\mathit{fst}(\mathit{rv})$ is checked until all candidates have been tried. We also do not give pseudocode for the procedure `abstract` called at line 23, but we will discuss in detail how abstractions are computed below (see page 116).

Need for propagation If a suitable abstraction a has been found, it must be added on top of the stack $C\ rv'$, becoming the actual configuration of rv'. According to the definition of abstraction, the set of program states represented by the new configuration of rv' is larger than the one of its previous configuration. Hence, replacing the configuration of rv' by a is not enough to guarantee that all feasible paths of $\mathcal{S}$ are preserved: there might exist black sub-paths that are feasible from rv but infeasible from rv'. After the abstraction, we want our analysis to consider those sub-paths as feasible from rv' since rv is now subsumed by rv'. Some part of the sub-tree rooted by rv' was computed before the abstraction, and configurations in that sub-tree have been obtained from the former configuration of rv': they might represent smaller sets of program states than expected and must be recomputed.

This can be done by applying symbolic execution again from the new

configuration of rv', adding the derived abstracted configurations on top of the stacks of configurations of descendants of rv' — we say that a is propagated from rv'. This can have two effects. First, the configuration propagated to a descendant might become satisfiable, while its previous configuration was not: such red vertices should be unmarked and pushed back on the stack *rvs* to be visited later (if they are not already in it). Second, red vertices to which abstractions are propagated might already be subsumed themselves. Propagating an abstraction to such a red vertex enlarges the set of program states represented by its configuration, which might cause the existing subsumption not to hold anymore, subsumption being defined as inclusion of sets of program states. This is also true for the subsumption that is currently attempted: the abstraction propagated to rv might not be subsumed by the abstraction computed at rv'. In such cases, there are two possibilities: deleting the previous subsumptions links to accept the new one (if the new configuration of rv allows it), or refusing the new subsumption link and stick with the existing ones. We have chosen the second way: the first one would require restarting the analysis from all discarded subsumees, while the second "only" requires to unfold the loop again from rv, searching for later subsumptions.

Note that abstracting the configuration of rv' only has an influence over the latter and its descendants, but has no effect on the feasibility of paths that do not go through rv'. Thus, there is no need to propagate a backwards, in a bottom-up manner.

Checking that the abstraction a can be safely propagated from rv' is done by the call to procedure `propagable` (line 24). If the check succeeds, a is actually propagated: this is done by the call to `propagate` (line 25), which is also responsible for actually pushing a on top of $C\ rv'$. Finally, the new subsumption link (rv, rv') is added to S (line 26) and subsumed returns *true*. How abstractions are performed, checked to be propagable and propagated is detailed below (see page 119).

Checking subsumption by constraint solving Checking subsumption of a configuration c by another c' is performed by a call to the constraint solver. This is done as introduced in Chapter 4: we check if the semantics of configuration c' is a logical consequence of the semantics of c. Given a configuration $c = (s, \pi)$ and a program state σ, the semantics of c, denoted sem_c, is the boolean expression that evaluates to *true* if and only if σ is a state of c, i.e.

$$\exists \sigma_{sym}.\ cons(\sigma, \sigma_{sym}, s) \wedge \pi(\sigma_{sym})$$

where $cons(\sigma, \sigma_{sym}, s)$ denotes the fact that σ and σ_{sym} are consistent with the store s, i.e. they associate the same values to program variables and their symbolic counterparts given by s. Subsumption of c by c' is established if the solver is able to prove that $\forall \sigma.\ sem_c(\sigma) \longrightarrow sem_{c'}(\sigma)$, i.e. it must

be proved that $sem_{c'}$ is valid assuming sem_c holds. Since SMT-solvers usually find satisfying assignments — or report that there are none — we actually ask the solver to prove that $\neg\, sem_{c'}$ is satisfiable: the subsumption is established only if the solver succeeds at proving the latter is false. The request passed to the solver is written in the SMT-LIB [9] format. It makes it possible to consider different solvers in an interchangeable manner in the implementation of our algorithm.

Abstracting Configurations

Abstracting a configuration consists in enlarging the set of program states it represents, i.e. in transforming it in another configuration that subsumes it. We recall the definition of the states of a configuration given in Chapter 3, subsumption being defined as the inclusion of such sets:

Definition 6. Let $c = (s, \pi)$ be a configuration. The *set of states of* c, noted $States(c)$, is the set $\big\{\sigma.\ \exists\sigma_{sym}.\ cons(\sigma, \sigma_{sym}, s) \wedge \pi(\sigma_{sym})\big\}$.

As shown by this definition as well as the definition of *cons*, the path predicate of a configuration shapes its set of states, the store merely being a link between program and symbolic variables. Abstracting a configuration is done by weakening its path predicate or, more precisely, by weakening its influence on the program variables. Below, we describe two ways for doing so. The choice of the method is controlled by a parameter and a unique method is used throughout an analysis.

The first is the most straightforward: simply weaken the path predicate itself. This is a plain loss of information and there are multiple ways to do so. In our case, we simply remove some constraints from the path predicate. Consider for example the configuration

$$c = \big(\{x \mapsto 0, b \mapsto 0\}, x_0 = 0 \wedge x_0 \leq b_0\big)$$

It represents the set of program states that associates x_0 to x, i.e. 0, and any positive value b_0 to b. Abstracting it by removing the first conjunct of its path predicate yields the configuration

$$c' = \big(\{x \mapsto 0, b \mapsto 0\}, x_0 \leq b_0\big)$$

which represents the set of states that associate any value x_0 to x, as long as this value is lesser or equal to b_0, the value associated to b.

The second consists in updating the index given by the store to some program variables. This has the effect of disconnecting such program variables from the constraints that weight over their previous symbolic counterpart,

but without actually removing these constraints from the path predicate. For example, abstracting x in c yields

$$c'' = \big(\{x \mapsto 1, b \mapsto 0\}, x_0 = 0 \wedge x_0 \leq b_0\big)$$

which represents the set of program states that associate any value x_1 to x and any positive value b_0 to b. Compared to the previous method of abstraction, we keep the fact that b must be positive, but loose that x must be less or equal than b. The fact that x was equal to 0 is forgotten in both cases.

Both methods have their pros and cons: the method that performs best depends on the program under analysis, as revealed by our experimental results. For example, the second method only disconnects some program variables from the constraints of the path predicate that weight over their symbolic counterpart and since these constraints might concern other program variables, the fact that they are not removed might lead to a better infeasible path elimination power than simply removing them. On the other hand, it can disconnect a program variable from too many constraints, forgetting some crucial information in the process, when simply removing constraints might have yield more accurate abstractions.

The crucial point is the choice of the constraints to remove or the program variables to update. Ideally, the combination of constraints to remove (resp. variables to update) that yields the most accurate abstraction w.r.t. infeasible paths elimination could be found, if it exists, by trying all possible combinations. Without surprise, this is impossible in practice, since the number of such combinations is exponential in the maximum number of elements to combine.

In this thesis we propose a simple algorithm for each method. It takes as inputs two configurations c and c' and a boolean expression ϕ, the safeguard condition of the red vertex of configuration c'. The goal is to find an abstraction a of c' that subsumes c and entails ϕ. Checking that ϕ is entailed by a is done in the same way as for subsumption. We will see in 5.3.4 that safeguard conditions are computed in a way that ensures that they are entailed by the configurations they label. This property must be preserved when abstracting a configuration. If a is rejected because it does not entail ϕ, we can also discard any abstraction of a, since this would enlarge its set of states even more and prevent entailment. On the opposite, if a vertex still has its default safeguard condition *true*, it is always possible to find a suitable abstraction of its configuration. The risk is to obtain a trivial abstraction in which all information carried by the path predicate has been forgotten.

Abstraction by constraint removal Removing constraints starts by deleting from the path predicate all occurrences of its first conjunct (path

predicates are implemented by lists of constraints), until a suitable abstraction of c' is obtained or until the current abstraction does not entail ϕ anymore.

Abstraction by store update Store update is performed analogously, i.e. by abstracting program variables one after the other until subsumption is established or entailment is lost. The collection of program variables of interest for the store update can be restricted as follows, depending on the fact that there are only subsumptions involving red vertices that occur along the same symbolic paths or not.

- In the case where subsumptions involve only configurations on the same symbolic path, let rv and rv' be the two red vertices associated with configurations c and c' respectively: rv is thus a descendant of rv'. Let p the path leading to rv. This path is of the form $pr \cdot su$, where pr is its prefix leading to rv' and su its suffix starting from the latter. All program variables that have neither been defined (i.e. the target of an assignment) nor abstracted along su correspond to the same symbolic expressions in rv and rv', since all potential modifications of these variables have been made along pr. Those program variables might be subject to more constraints at rv than at rv', since constraints might have been added during symbolic execution along su. This is not a problem: the path predicate at rv entails the path predicate of rv', since in abstraction by store update, constraints are never removed from path predicates. The only program variables that are worth abstracting are those that were defined or abstracted along su. Those variables are exactly those whose indexes as given by the stores of configurations of rv' and rv differ.

- In the case where rv and rv' occur along different symbolic paths, let p and p' be the paths leading to rv and rv', respectively. Since R is a tree, p and p' have a longest common prefix that we call pr: p and p' are of the form $pr \cdot su$ and $pr \cdot su'$, respectively. It is not needed to abstract those variables that were neither defined nor abstracted along su or su' (they correspond to the same symbolic expressions at both rv and rv'), but variables that were defined or abstracted along at least one of each suffix might be worth abstracting. This is not sufficient however to guarantee that there is an abstraction of $\texttt{top}(C\ rv')$ that subsumes $\texttt{top}(C\ rv)$. Consider the following case, where su and su' are each made of a single transition, both labeled by a guard, for example Assume $x < y$ and Assume $\neg(x < y)$, respectively. No program variables are defined or abstracted along su or su', thus all program variables correspond to the same symbolic expression at rv and rv'. However, the configuration of rv entails $x < y$ while the

configuration of rv' entails $\neg(x < y)$, preventing any subsumption of rv by rv' if x or y (or both) are not abstracted. To sum up, in the case of subsumptions across different symbolic paths, the variables worth abstracting are those: *(i)* that are defined or abstracted or *(ii)* that are used in guards along the suffixes leading to rv and rv'. Variables that are defined or abstracted along su (resp. su') are exactly those whose indexes as given by the store of the configuration of rv (resp. rv') and by the configuration of the vertex at the end of pr differ.

We call *abstractable variables* the variables worth abstracting. Some of them can be discarded as interesting targets for store update: variables that are already abstracted at the potential subsumer rv', and those that are not used or assigned along the path leading to rv'. In both cases their symbolic counterparts do not occur in the path predicate of the latter and can take any value. They do not influence the set of program states of the configuration of rv' and can be considered abstracted at this point.

These two methods for abstracting configurations are simple and greedy, but they have the advantage of only requiring a number of calls to the solver that is linear in the number of constraints in the path predicate (or program variables for store updates).

Propagating Abstractions

We now describe how abstractions are propagated, and how we check in the first place if they can actually be safely propagated. We start by describing the latter, which we believe will help understand the former.

Checking propagability Deciding if the abstraction can be propagated consists in propagating it a first time while checking that it does not cancel any existing subsumption link and, if the two candidates for subsumption lie on the same path, that the new subsumption still holds after the propagation. The procedure `propagable` responsible for doing so is described in Algorithm 4. It takes as inputs the red vertex rv whose configuration has been abstracted, the abstraction a that is to be propagated, the subsumption link sub that we currently try to establish and the red-black graph RB. The idea is to check if the actual configuration of rv can safely be replaced by a. This depends, for an arbitrary rv — typically the argument of one of the recursive calls to `propagable` (see line 40), on the fact that rv is marked, subsumed, or neither.

If rv is marked (line 29), its configuration can always be replaced since rv cannot be the source of a subsumption. If its new configuration a becomes satisfiable, due to the abstraction, rv must be unmarked and pushed back on the stack of vertices to visit rvs. In practice this is handled as part of the

Algorithm 4: `propagable`

```
   inputs : a configuration a, a red vertex rv, a subsumption sub, a red-black
            graph RB
   output: a boolean value
29 if M rv then
30 |  return true;
31 if subsumer(rv) = Some(rv') then
32 |  return a ⊑ top(C rv');                                  /* CS */
33 if rv = fst(sub) then
34 |  return a ⊑ top(C snd(sub));                             /* CS */
35 if rv ∈ LH then
36 |  a ← combine_abs(top(C rv), a);
37 push(a, C rv);
38 foreach δ ∈ Δ_o(rv) do
39 |  let a' = SE top(C rv) label(e);
40 |  if ¬ propagable(a', tgt(e), sub, RB) then
41 |  |  pop(C rv);
42 |  |  return false;
43 pop(C rv);
44 return true;
```

actual propagation of a, once all recursive calls to `propagable` succeed and guarantee that the abstraction is propagable everywhere in the sub-tree.

If rv is already subsumed (line 31), the algorithm checks if a is subsumed or not by the configuration of the subsumer of rv (line 32): if this is the case, the actual configuration can be replaced by a without invalidating the subsumption link starting at rv. Otherwise the abstraction is deemed incompatible with the subsumption.

If rv is neither subsumed nor marked, the algorithm checks that it is the red vertex that it actually attempted to subsume (line 33). If this is the case, the attempted subsumption must hold despite the loss of information due to the propagation of the abstraction (line 34).

If rv is not the vertex of the attempted subsumption, the algorithm checks that it is an occurrence of a loop header in the input LTS (line 35). If this is the case, then its configuration might have been abstracted previously: the new abstraction should reflect the loss of information performed earlier and the one induced by the propagated abstraction a. At line 36 a new version of a is computed that combines both abstractions. We describe later how this combination is realized — it depends on which of the two methods of abstraction is used, but in both cases it results in a configuration that subsumes both the previous configuration of rv and a. If the previous configuration has not been abstracted previously, `combine_abs`

simply returns a.

We could have chosen to build the combination of a and `top`($C\ rv$) only if `top`($C\ rv$) $\not\sqsubseteq a$, i.e. if an abstraction looser than a has been previously performed at rv. We rather chose to systematically combine configurations at occurrences of loop headers: this is not a problem since `combine_abs` simply returns a if no abstraction was performed at rv. This also saves some calls to the solver since `combine_abs` proceeds without calling it.

The abstraction a (or the result of its combination with previous abstractions) is then pushed on top of $C\ rv$ (line 37), before being propagated by symbolic execution to each successor of rv through the recursive calls to `propagable` (line 40) and the global result sent to the caller. Before returning, a is always popped from $C\ rv$ (lines 41 and 43): at this point, we only want to know if a can be propagated in this sub-tree. The actual propagation is done in a second pass, once the abstraction has been checked to be globally propagable.

Propagating abstractions If the call to `propagable` globally succeeds, the (original) abstraction a is actually propagated from rv. This is done by the call (line 25 in Algorithm 3) to procedure `propagate` whose code is given in Algorithm 5, which is very similar to `propagable` and proceeds as follows.

Algorithm 5: `propagate`

inputs : a configuration a, a red vertex rv, a red-black graph RB
output: none

```
45 if M rv then
46 |   push(a, C rv);
47 |   if sat(top(C rv)) then                                   /* CS */
48 |   |   M → M(rv := false);
49 |   |   Φ → Φ(rv := true);
50 |   |   if rv ∉ rvs then
51 |   |   |   push(rv, rvs);
52 else if subsumer(rv) = Some(rv') then
53 |   push(a, C rv);
54 else
55 |   if rv ∈ LH then
56 |   |   a ← combine_abs(top(C rv), a);
57 |   push(a, C rv);
58 |   foreach e ∈ Δ_o(rv) do
59 |   |   let a' = SE top(C rv) label(e);
60 |   |   propagate(a', tgt(e), RB);
```

If (an arbitrary red vertex argument) rv is marked, the propagated abstraction a is pushed on top of the current configuration stack, the algorithm checks for its satisfiability and, on a positive answer, rv is unmarked in M and pushed back on rvs (if not already occurring in it) while its safeguard condition is restored to $true$ (line 45 to 51). If rv is subsumed, a is simply pushed on top of $C\ rv$ (lines 55 and 51). If rv is neither marked nor subsumed, we proceed as for `propagable`: we push a (or the combination of a and $\texttt{top}(C\ rv)$ if rv is a loop header) on top of $C\ rv$ and propagate it by symbolic execution to successors of rv, which are then processed through recursive calls (line 55 to 60). When said calls end, a is not popped from $C\ rv$, unlike in `propagable`.

Combining Abstractions

The way abstractions are combined depends on the method used to abstract configurations. We recall that methods of abstraction are not mixed during an analysis. With both methods the idea is the same: given a red vertex to which an abstraction is propagated, we want to build a configuration that reflects the losses of information induced by the propagated abstraction and previous abstractions that might already exist at this vertex.

Let us call rv' the red vertex from which an abstraction a' is propagated, and rv a descendant of rv' that is an occurrence of some loop header (not necessarily the same loop header as the one represented by rv') to which a' has to be propagated. By iterating propagation of abstraction between successive loop headers along the path, we can assume without loss of generality that there is no other occurrence of a loop header between rv' and rv: a' and rv' might not be the initial abstraction and root at the start of a propagation, but some intermediate step of a larger propagation.

Let sp be the red sub-path leading from rv' to rv and a the abstraction obtained from a' that is supposed to become the new configuration for rv : we have $a = SE^*\ trace(sp)\ a'$.

Let us consider first the case where no abstraction took place along the path leading from rv' to rv. Then, there must exists a configuration c' in the stack of configurations of rv' such that $\texttt{top}(C\ rv) = SE^*\ trace(sp)\ c'$. By definition of a', we have that $\texttt{top}(C\ rv') \sqsubseteq a'$ and thus $\texttt{top}(C\ rv) \sqsubseteq a$ since SE^* is monotonic for subsumption. Thus, a can simply be pushed on top of $C\ rv$.

Let consider now the case where some abstraction was performed along the path leading from rv' (excluded) to rv. It is now most certainly not the case that $\texttt{top}(C\ rv) = SE^*\ trace(sp)\ c'$ since abstractions have been either performed at or propagated to rv. As a result, it is usually not the case that the current configuration of rv is subsumed by the propagated abstraction a. The algorithm must compute an abstraction of a that subsumes the current configuration of rv.

Combining abstractions obtained by constraint removal We first consider abstractions that are performed by removing constraints from path predicates. Since stores are not modified in this case, those of $\texttt{top}(C\ rv)$ and a must be identical, however their path predicates contain only (most probably different) subsets of the conjuncts of the path predicate of c. The combination of the current configuration of rv and a is the configuration whose store is identical to those of $\texttt{top}(C\ rv)$ and a and whose path predicate contains exactly the constraints that occur in the path predicates of both $\texttt{top}(C\ rv)$ and a.

Combining abstractions obtained by store update Let us consider now abstractions that are performed by updating stores. Stores of $\texttt{top}(C\ rv)$ and a might now be different, and conjuncts of the path predicate of a might not be a subset of those of c anymore, since a has been obtained by SE^* from a', whose store is an updated version of the store of c'. This is not a problem: the important fact is that a number of program variables were abstracted at rv and those abstractions must be reflected in its new configuration. The configuration that is pushed on top of $C\ rv$ is obtained from a by abstracting those program variables that were abstracted in $\texttt{top}(C\ rv)$, i.e. the variables whose symbolic counterparts, as given by the store of $\texttt{top}(C\ rv)$, do not occur in the path predicate of the latter.

For example, let $a = \big(\{x \mapsto 2, y \mapsto 1\},\ x_2 \geq 0\ \wedge\ y_1 = y_0 + 1\big)$ and $\texttt{top}(C\ rv) = \big(\{x \mapsto 1, y \mapsto 2\},\ x_1 \geq 0 \wedge y_1 = y_0{+}1\big)$. Program variable y can be considered abstracted in the latter, since its symbolic counterpart does not occur in the path predicate of the configuration of rv. The combination of these two configurations should reflect the loss of information performed at rv by updating y and is obtained by simply abstracting the latter in a: the configuration pushed on top of $C\ rv$ is $\big(\{x \mapsto 2, y \mapsto 2\},\ x_2 \geq 0\ \wedge\ y_1 = y_0 + 1\big)$.

In Chapter 4 we did not provide a proof that our methods of combination actually yield configurations that subsumes both of their entries: transformations were supposed to be applied in the right order and there was no need there to consider propagation, and thus combination issues. The formalization introduced in Chapter 4 could be extended to prove that the combination of two configurations c and c' is a configuration that subsumes both c and c'. The proof is trivial for the case of constraints removal: it is a direct consequence of definitions of subsumptions, set of program states, etc. The case of store updates is a bit less trivial: proving it within Isabelle/HOL would be tedious because of the handling of the indexes of symbolic variables.

5.3.4 Refine-and-Restart Mechanism

We now describe the implementation of the counterexample guided refinement mechanism mentioned previously. As said in Section 5.3.2, lines 9 to 14 of the main procedure `build` are responsible for this part of the analysis, granted that this mechanism was enabled through the global boolean flag *restart*. Here, we consider the case where rv is a non-marked occurrence of a final location of $\mathcal{S}$: the algorithm checks whether the path of R (unique, without traversing subsumption links) leading to rv is genuinely feasible from the initial configuration c_i, or if its infeasibility has not been detected because of some abstraction that occurred along that path. If the latter is true, a safeguard condition ϕ is computed that captures the reasons of the infeasibility of that given path, and the red vertex where the faulty abstraction occurred is labeled with ϕ, blocking this abstraction. The analysis is restarted from there by unfolding the loop: subsumptions will have to happen at some descendants, if ever. We detail the whole process below.

Detecting faulty abstractions First, the path p leading to rv (line 6 in Algorithm 2) is retrieved following the transitions in the tree. If p is infeasible from the initial configuration c_i (line 7), then its infeasibility is not due to an abstraction introduced along p. Otherwise, procedure `faulty_abs` called at line 8 is devoted to find the first red vertex along p where was performed an abstraction that causes the infeasibility of p to be unnoticed, i.e. the first red vertex along p where an infeasible suffix of p was turned into a feasible one. The process is simple: it is a search for the first occurrence rv' of a loop header traversed by p whose stack of configurations contains two successive configurations c_m and c_n (c_n being on top) such that the suffix of p starting at rv' is infeasible from c_m but feasible from c_n: c_n is the faulty abstraction.

Computing safeguard conditions Once such rv' has been found, a safeguard condition is computed by a weakest precondition calculus along the infeasible suffix of p, and rv' is labeled with it (line 9). Our notion of weakest precondition calculus is given in the following definition.

Definition 22. Given a label $l \in$ *Labels* and a boolean expression over program variables ϕ, the weakest precondition of ϕ w.r.t. l is given by the function *WP*:

$$WP\ l\ \phi = \begin{cases} \phi & \text{if } l = \text{Skip} \\ \neg\,\phi' \ \vee\ \phi & \text{if } l = \text{Assume } \phi' \\ \phi\{v \mapsto e\} & \text{if } l = \text{Assign } v\ e \end{cases}$$

where $\phi\{v \mapsto e\}$ denotes the substitution of occurrences of v by e in ϕ.

We extend WP to sequences of labels as follows:

Definition 23. Given a sequence of labels ls and a boolean expression over program variables ϕ, the weakest precondition of ϕ along ls is given by the function WP^*:

$$WP^*\ ls\ \phi = \begin{cases} \phi & \text{if } ls \text{ is empty} \\ WP^*\ ls'\ \big(WP\ l\ \phi\big) & \text{if } ls \text{ is of the form } ls' \cdot [l] \end{cases}$$

The exact safeguard condition used to label rv' is the conjunction of its previous safeguard condition with the weakest precondition of *false* along the shortest infeasible suffix of p starting at rv'. Indeed, the purpose of a safeguard condition is to remember some crucial information regarding the infeasibility of a given (set of) sub-path(s) starting at red vertex rv': it must partially describe the set of program states of the current configuration of rv', i.e. it must be entailed by this configuration.

The goal is to prune as many infeasible paths as possible: this condition must to be as strong as possible to better over-approximate the set of feasible sub-paths starting at rv'. Since the configuration of the red vertex in which p ends is unsatisfiable, its set of states is empty. The strongest condition that is entailed by this configuration is thus *false*.

Weakest precondition calculus has the following property: given a configuration c, a label l and a condition ϕ over program variables such that $SE\ c\ l \models_c \phi$, we have that $c \models_c WP\ l\ \phi$. This extends to WP^* and sequences of labels. This property and the fact that we only consider abstractions that preserve entailment guarantee that red vertices are always labeled with safeguard conditions that are entailed by their current configurations. This property and its equivalent for WP^* have been proved in our formalization (see A.6.10).

Restarting the analysis Once rv' has been labeled with its safeguard condition, its configuration prior to the faulty abstraction is restored, i.e. the elements from its stack of configurations are removed until c_m, the last known configuration from which the infeasibility of p can be detected, is on top. Then, the sub-graph starting at rv' is destroyed. The intermediate red-black graph RB is updated in the following way:

- configuration stacks of descendants of rv' are removed from C,
- subsumption links involving rv' or one of its descendants are removed from S; subsumees of descendants of rv' that are not descendant of rv' themselves must be pushed back onto rvs (so that they can be subsumed once again or their descendants be built),
- descendants of rv' are unmarked in M,

- edges of R going from or to descendants of rv' are removed from E,
- descendants of rv' are removed from V and from the stack of vertices to visit rvs.

Finally, rv' is pushed back on rvs so that the analysis restarts from there.

Non-termination issue The two methods of abstraction introduced earlier do not learn from safeguard conditions, i.e. when the configuration of a red vertex is to be abstracted, it does not matter if this vertex is labeled with a safeguard condition or not: the same sequence of abstractions is computed, regardless of the presence of the condition. Labeling a red vertex with a safeguard condition simply prevents to establish the subsumption that yielded the faulty abstraction of rv' and forces the descendants of rv to be built. In the best cases, this will lead to discovering more accurate subsumption links starting from these descendants. In the worst cases, this might only postpone the faulty abstraction, which will occur again, but between different occurrences of the loop header represented by rv and rv'. This will in turn introduce the same infeasible paths once again and the abstraction will be refined again later, and so on. This might yield to an infinite chain of refine-and-restart phases, preventing the algorithm to terminate.

This problem can be mitigated by bounding the length of red paths through the global parameter *mrl*, or by simply disabling the counterexample mechanism through the boolean flag *restart*. In the first case, the final LTS will be obtained by plugging the partial red part of RB to its black part. The accuracy of the result w.r.t. to feasible paths increases with *mrl*: the set of feasible paths of length *mrl* or lesser might be fairly well approximated, but the accuracy of the result dramatically decreases when considering longer paths that end in the black part.

For the moment, safeguard conditions computed when refine-and-restart phases occur are our only way to prevent some abstractions. Disabling the counterexample guided refinement mechanism yields LTS that are complete unfoldings of the input ones but since in this case abstractions are never restricted through safeguard conditions, the resulting LTS might contain a lot or all the infeasible paths of the original LST. We introduce and describe below a mechanism based on some form of "look-ahead" for preventing too crude abstractions.

5.3.5 Look-Ahead Mechanism

The counterexample guided refinement introduced above has two drawbacks which we already mentioned. First, it can cause the algorithm to not terminate if a chain of refine-and-restart occurs. In such cases, we only have the choice between: *(i)* either force the algorithm to halt which requires plugging the partial red part into the black one or *(ii)* disable the coun-

terexample guided refinement. Second, its infeasible path elimination power is limited by the fact that we only search for faulty abstractions along the shortest paths leading to final occurrences.

We now introduce a different mechanism for enhancing the infeasible paths elimination power of our algorithm. Unlike the refine-and-restart mechanism, we prevent some abstractions before they are performed. This is done, given a red vertex *rv* to subsume, by restricting the set of potential subsumers among the previous occurrences of *fst*(*rv*) based on similarities between sets of feasible paths starting at both occurrences. We call *look-ahead* this mechanism, since it basically consists in catching a glimpse of the near futures of the potential subsumees and subsumers.

Subsumption was defined in Chapter 3 as an inclusion of sets of program states, which entails an inclusion of sets of feasible paths between the subsumee and the subsumer. In the context of infeasible path pruning, an ideal theoretical definition for subsumption would require the sets of feasible paths starting at both candidates to be equal. This requirement is far too strong in practice: it is usually impossible to build such sets, and this is actually the problem we are trying to tackle in the first place by over-approximating the set of feasible paths of the input LTS.

Simply checking for subsumption as presented earlier might be too liberal and building full sets of feasible paths starting at the subsumee and at the subsumer is usually impossible. However, we can reasonably check that these sets are equal only up to a certain depth. The underlying idea is that the more similar the sets of feasible paths starting at the potential subsumee and subsumer, the less it will be needed to abstract the current configuration of the latter. When trying to subsume a red vertex *rv*, we check subsumption and search an abstraction, for each potential subsumer rv', only if the sets of feasible paths of length lesser or equal to *la* (which stands for look-ahead, our final global input parameter) starting at *rv* and rv' are equal. This is done by the call to `compare_fp_sets` at line 19 in procedure `detect_sub` (see Algorithm 3). If feasible paths sets are found to differ, candidate rv' is immediately discarded. The expected effect is that the algorithm is driven to only consider the potentially more accurate subsumptions.

This mechanism only gives hints. First, there is no guarantee that restricting candidate subsumers will actually lead to more subtle abstractions: since abstraction methods remove constraints or update stores in a given order, it is possible that some crucial information w.r.t infeasible path detection is still lost during abstraction. Second, we compare sets of feasible paths starting from both red vertices before actually performing the abstraction: sets of feasible paths might be equal up to a certain depth at that moment but a later step of abstraction at the subsumer might enlarge its set of states, thereby breaking the previous equality of sets of feasible descendants. Even if the equality between sets of paths starting at the subsumer and subsumee still holds, those sets might simply have changed after

performing and propagating abstractions, since this can caused vertices to be unmarked. Nonetheless, experimental results presented in 6.1 show that the number of infeasible paths pruned from input LTS greatly increases in most cases with appropriate values of la.

5.3.6 Building The Resulting LTS

Once the analysis is over, RB is turned back into a new LTS $\mathcal{S}'$ by removing from the red part R the edges leading to marked red vertices, replacing the targets of edges leading to subsumed red vertices by their subsumers, then renaming vertices and label edges between red vertices with the label of the edge between their black counterparts. For red vertices where the analysis halted because of the mrl limit, if any, the edges whose target is not final are connected to the corresponding vertex in the black part, i.e. to the original CFG. This trick and the fact that transformations on the red part never rule out (prefixes of) feasible paths ensures that $\mathcal{S}'$ preserves the feasible paths of $\mathcal{S}$. The set of paths of the resulting LTS is exactly the set of red-black paths of the intermediate red-black graph.

5.4 The Merging Sort Example

5.4.1 The Merging Sort Program

In this section, we illustrate how our algorithm behaves on the merging sort program, whose pseudocode and LTS are shown in Figure 5.1 and Figure 5.2, respectively.

The `merge` function takes as inputs two sorted arrays of integers a and b, their respective lengths la and lb and a third array T in which elements of a and b will be stored in the ascending order. It is basically made of three loops. The first loop — the main one — iterates over elements of a and b and stores them in T depending on their values. When both a and b are initially empty, the main loop is not entered, and neither are the last two loops. Otherwise, when the execution exits this first loop, it is always the case that exactly one array, a or b, has been completely visited: the two secondary loops are there to store in T the elements of the array that has not been entirely processed through iterations of the first loop.

Although this program is simple, its LTS presents a great number of infeasible paths. Their infeasibility is due to the dependencies between iterations in the first loop and the traversals of the two remaining loops: the transition used to exit the first loop controls which of the two remaining loops is traversed, and all this depends on which index was incremented last in the first loop. This is also the reason for the second and third loop to be exclusive. We identify seven "groups" of infeasible paths:

```
Function merge(int[] a, int[] b, int la, int lb, int[] T)
1     let ia = 0;
2     let ib = 0;
3     while ia < la do
4         if ib < lb then
5             if a[ia] < b[ib] then
6                 T[ia + ib] ← a[ia];
7                 ia ← ia + 1;
              else
8                 T[ia + ib] ← b[ib];
9                 ib ← ib + 1;
          else
10            break;
11    while ia < la do
12        T[ia + ib] ← a[ia];
13        ia ← ia + 1;
14    while ib < lb do
15        T[ia + ib] ← b[ib];
16        ib ← ib + 1;
```

Figure 5.1: The merging sort program.

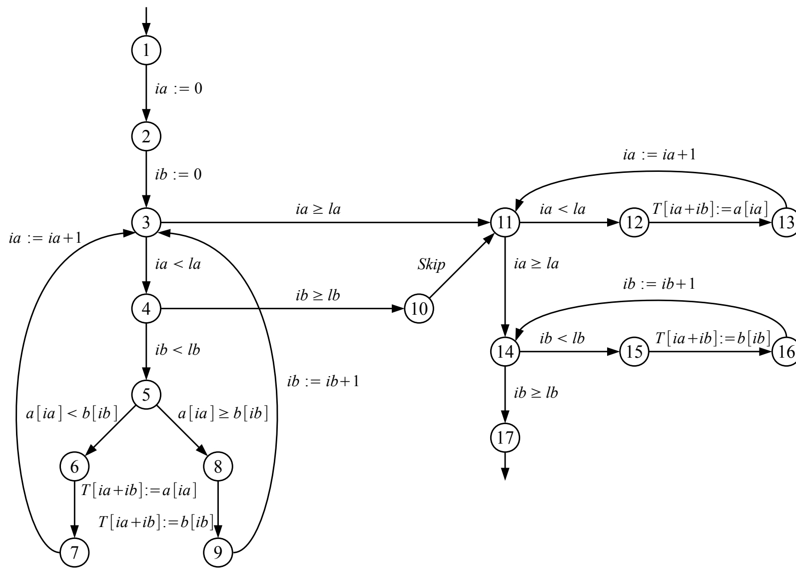


Figure 5.2: The merging sort LTS.

1. paths that exit the first loop by the transition going from 3 to 11 and enter the second loop,
2. paths that exit the first loop by the transition going from 4 to 10 and enter the third loop,
3. paths that enter the first loop, then exit it by the transition going from 3 to 11 and that do not enter the third loop,
4. paths that exit the first loop by the transition going from 4 to 10 and that do not enter the second loop,
5. paths that exit the first loop by the transition going from 4 to 10 immediately after going through transition from 7 to 3,
6. paths that exit the first loop by the transition going from 3 to 11 immediately after going through transition from 9 to 3,
7. paths that go through both the second and the third loops.

These groups are not disjoint: paths might be infeasible for several reasons. For example, paths that exit the first loop through the transition going from 3 to 11 and enter both the second and third loops are elements of groups 1 and 7 at least.

One can note that, in this particular example, the infeasibility of paths is never directly related to the values of the elements of input arrays a and b, but only to the fact that paths go through inconsistent sequences of assignments/guards regarding indexes ia and ib. This can be interpreted as the fact that, for every feasible path in the merging sort LTS, one can always find a valuation of the elements of a and b that is consistent with this particular path. For this reason, we illustrate how our algorithm behaves on a slightly simplified version of the LTS depicted in Figure 5.2 in which we remove transitions that use the values of the elements of a and b. This simplified LTS is shown in Figure 5.3. Removing these transitions from the LTS of Figure 5.2 does not impact the accuracy of the results produced by our algorithm. Discovering such abstraction of the program would rely on applying techniques from data flow analysis, on the expressions that guard loops and conditionals.

In Figure 5.3, since 5 is a branching vertex, its two outgoing edges should have been labeled by a condition: the latter would have been *true* on both edges because of the abstraction of the original comparison between arrays elements and the incrementation of the corresponding index would label the next edge (the vertices for the assignment to T being removed). To alleviate the description of the example, we rather put the incrementations on the edges leaving location 5. In Chapter 3, we worked with LTS where it exists only one transition linking two distinct locations, and this is also a requirement in our current implementation. Thus, we add two transitions labeled by Skip going respectively from 6 and 7 to 3 in order to avoid having

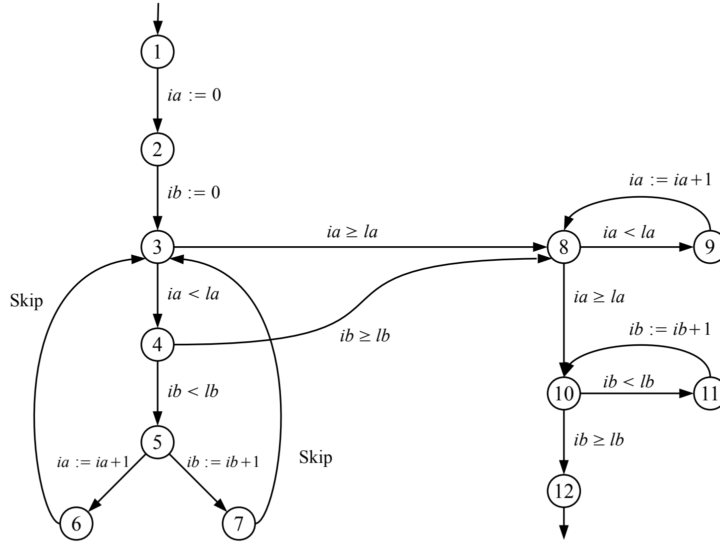


Figure 5.3: Simplified version of the merging sort LTS.

two transitions linking 5 to 3. Finally, the unnecessary *Skip* edge from 10 to 11 in the input LTS has also been removed.

In the following, we use this simplified example to show how our algorithm behaves with different values of its control parameters. In each of the following cases, we do not consider any user-given precondition: the initial configuration is always $c_0 = \big(\{ia \mapsto 0, ib \mapsto 0, la \mapsto 0, lb \mapsto 0\}, true\big)$. Moreover the global parameter *dp* is set to *false* and we only consider subsumption links that involve red vertices that occur along the same symbolic paths, i.e. the potential subsumees are always descendants of their potential subsumers. We will discuss the case of subsumptions between different symbolic paths in the following chapter.

5.4.2 Merging Sort without Path Sets Comparisons

Abstraction by Constraint Removal

In this section, we consider that abstractions are performed by removing constraints from path predicates and that the counterexample guided refinement mechanism is enabled. Our goal here is to illustrate how the basic features of our algorithm interact and, for the sake of simplicity, we suppose that sets of feasible paths starting at the potential subsumees and subsumers are not compared when trying to establish subsumptions (i.e. the global parameter *la* is set to 0). As a result, the infeasible path detection power of the algorithm in this particular case is somewhat limited, and we will see that the resulting LTS retains most of the infeasible paths of the input one.

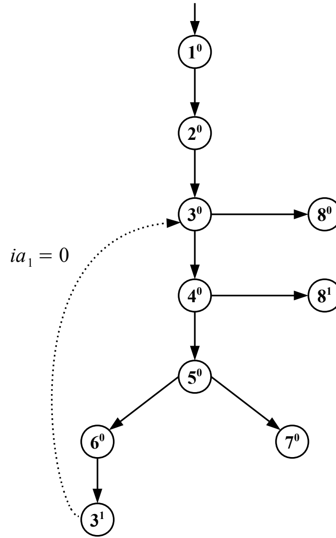


Figure 5.4: Building the first path in the body of the main loop.

Note however that this is only for illustrative purposes and we will see in the following that the infeasible path detection power is greatly improved with a suitable value for la.

The analysis starts at the first occurrence 1^0 (see Figure 5.4) of the initial location of the LTS of Figure 5.3. The superscripts at program location reflect the order of visit.

Handling the first assignments (Fig. 5.4) The algorithm starts by applying two steps of symbolic evaluation: 2^0, the successor of 1^0, is built since 1^0 is not marked, not an occurrence of a final black vertex and cannot be subsumed since it is the only of occurrence of black vertex 1. The same case applies for adding the edge between 2^0 and its only successor 3^0.

Execution of the body of the first loop (Fig. 5.4) The two successors of 3^0, 4^0 and 8^0, are built. Let us say that the latter is pushed first on the stack of vertices to visit: the next vertex to visit is thus 4^0. The constraint solver is called to check the satisfiability of configurations of both 4^0 and 8^0: none of them is marked, since their configurations can be proved to be satisfiable. Successors 5^0 and 8^1 of 4^0 are built and the solver is called to check satisfiability of their configurations: none of them is marked. Then let us say that 8^1 is pushed first on the stack of vertices to visit: the next vertex to be processed is 5^0. Once again, its successors are built: we consider that 7^0 is pushed first and that the analysis goes on at 6^0, whose unique successor 3^1 is built and processed.

Subsumption of 3^1 by 3^0 (Fig. 5.4) Since an occurrence of a loop header has been reached, the algorithm attempts to establish the subsumption of 3^1 with the only existing occurrence of 3, namely 3^0. The configuration of 3^1, denoted c_{3^1} is

$$\begin{aligned} c_{3^1} = \big(&\{ia \mapsto 2, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, \\ &ia_1 = 0 \wedge ib_1 = 0 \wedge ia_1 < la_0 \wedge ib_1 < lb_0 \wedge ia_2 = ia_1 + 1\big) \end{aligned}$$

which is not subsumed by the current configuration of 3^0

$$c_{3^0} = \big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ia_1 = 0 \wedge ib_1 = 0\big)$$

since the latter requires ia to be 0 while the former requires it to be 1. The algorithm starts searching for an abstraction of c_{3^0} that both subsumes c_{3^1} and entails the current safeguard condition of 3^0, which is simply *true* since no refine-and-restart phase has been triggered for now. By removing from the path predicate of c_{3^0} the unique occurrence of its first conjunct $ia_1 = 0$, one obtains the following abstraction a of c_{3^0}:

$$a = \big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ib_1 = 0\big)$$

which happens to subsume c_{3^1} (and entail *true*). Before pushing a on top of the stack of configurations of 3^0 one first has to check that it can safely be propagated. Since no subsumption links start from the current descendants of 3^0, this comes down to check that the abstraction propagated by symbolic evaluation up to 3^1 is actually subsumed by a. The configuration that is propagated to 3^1 is

$$\begin{aligned} a' = \big(&\{ia \mapsto 2, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, \\ &ib_1 = 0 \wedge ia_1 < la_0 \wedge ib_1 < lb_0 \wedge ia_2 = ia_1 + 1\big) \end{aligned}$$

which happens to be subsumed by a: the latter is pushed on top of the stack of configurations of 3^0 and is propagated from there. Finally, subsumption $(3^1, 3^0)$ is added to the subsumption relation (see 5.5).

Subsumption of 3^2 by 3^0 (Fig. 5.5) The analysis resumes at the vertex on top of the stack of vertices to visit: 7^0, whose only successor, 3^2 is built. Once again, an occurrence of a loop header has been reached, and the algorithm attempts to establish the subsumption of 3^2 by a previous occurrence of black vertex 3. Since 3^1 is already subsumed and lies on a different path than 3^2, the only potential subsumer is 3^0 again. The configuration at 3^2 is

$$\begin{aligned} c_{3^2} = \big(&\{ia \mapsto 1, ib \mapsto 2, la \mapsto 0, lb \mapsto 0\}, \\ &ib_1 = 0 \wedge ia_1 < la_0 \wedge ib_1 < lb_0 \wedge ib_2 = ib_1 + 1\big) \end{aligned}$$

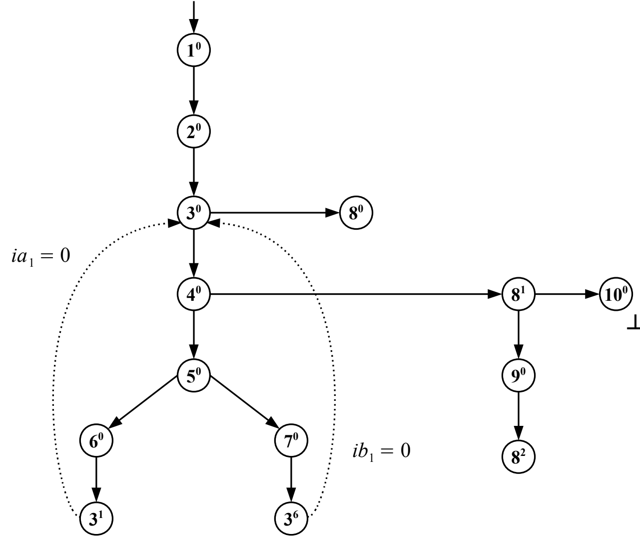


Figure 5.5: Both paths in the main loop have been subsumed. The algorithm attempts at subsuming 8^2.

which is not subsumed by the current configuration of 3^0, since both require ib to have different values (0 at 3^0, 1 at 3^2). The algorithm proceeds as previously: it removes from the path predicate of c_{3^0} its first conjunct, which is now $ib_1 = 0$. Thus one obtains the following abstraction a of c_{3^0}:

$$a = \big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, true\big)$$

which subsumes c_{3^2} and can be safely propagated from 3^0: the abstraction propagated to 3^1 is actually subsumed by a, as well as the one propagated to 3^2. Configuration a is pushed on top of the stack of configurations of 3^0 and propagated from there. The path predicate of the configuration of 3^0, and its two previous configurations are stored in its stack.

Subsumption of 8^2 by 8^1 (Fig. 5.5 and 5.6) The next red vertex to visit is 8^1. Since there exist no other occurrences of black vertex 8, its successors 9^0 and 10^0 are built and pushed on the stack of vertices to visit. The configuration at 10^0 requires ia to be both lesser and greater or equal to la, which is impossible: we suppose that the solver in use detects the infeasibility of the configuration of 10^0 and that the latter is marked (with a $\perp$ mark in the figures). Let us say that 10^0 is pushed first and that the analysis resumes at 9^0. The only successor of the latter, 8^2, is built, pushed on top of the stack of vertices to visit and then processed. The algorithm attempts to subsume 8^2 with a prior occurrence of 8 along the current path,

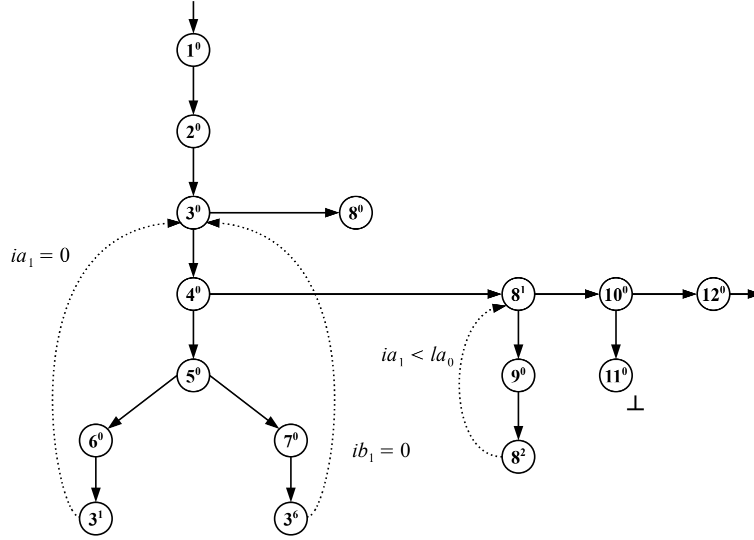


Figure 5.6: The first final red vertex has been reached: the algorithm searches for a too crude abstraction.

i.e. 8^2. The configuration of 8^2 is

$$c_{8^2} = \big(\{ia \mapsto 2, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\},\\ ia_1 < la_0 \wedge ib_1 \geq lb_0 \wedge ia_2 = ia_1 + 1\big)$$

which is subsumed by the following abstraction of c_{8^1}:

$$a = \big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ib_1 \geq lb_0\big)$$

which is in turn safely propagable from 8^1. This abstraction is pushed on top of the stack of configurations of 8^1 and propagated from there. During the propagation of a from 8^1, the current configuration of 10^0 becomes

$$\big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ib_1 \geq lb_0 \wedge ia_1 \geq la_0\big)$$

which is no longer unsatisfiable: 10^0 is unmarked (but not pushed on the stack of vertices to visit: it is already in it).

Towards the first final red vertex (Fig. 5.6) The analysis resumes at 10^0. Since 10^0 is unmarked and is the first occurrence of black vertex 10, its successors 11^0 and 12^0 are built and pushed on top of the stack of vertices to visit. When it is built, 11^0 is marked, since its configuration requires *ia* to be both greater or equal and lesser than *la*. Since, it is marked, no particular treatment is applied to 11^0 and the analysis resumes at 12^0. Note the arrow going out of 12^0: it indicates that the latter is an occurrence of a final black vertex, although we do not consider graphs to have final vertices.

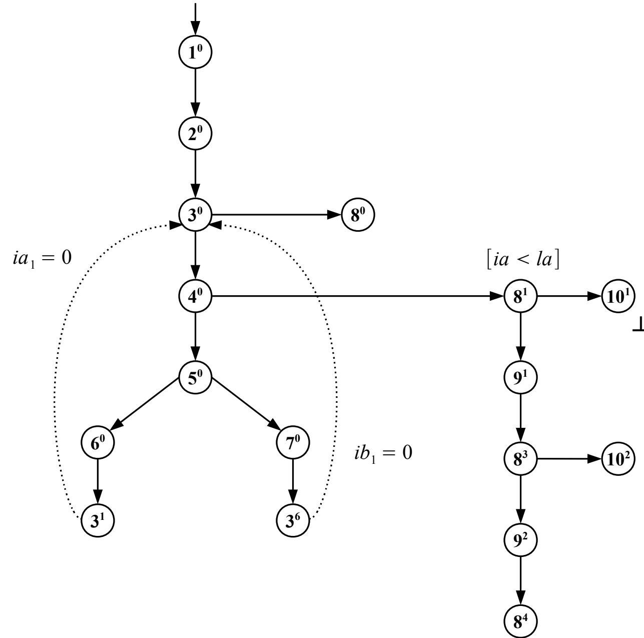


Figure 5.7: Refining the faulty abstraction at 8^1 and restarting the analysis causes the second loop to be unfolded twice.

Refine the faulty abstraction (Fig. 5.6 and 5.7) The latter is a non-marked occurrence of a final black vertex: the algorithm now searches for an abstraction along the shortest path from 1^0 to 12^0 that potentially made that path feasible when it was infeasible in the first place. Indeed, the abstraction performed at 8^1 made the path feasible by unmarking 10^0: a refine-and-restart phase is triggered. Red vertex 8^1 is labeled with the weakest precondition of *false* along $8^1 \cdot 10^0$, the shortest infeasible prefix of the sub-path starting at 8^1, i.e. $ia < la$. Its configuration prior to the faulty abstraction is restored, its sub-graph destroyed, and 8^1 is pushed back on the stack of vertices to visit, and selected during the next iteration of `build`.

Restart the analysis (5.7) The analysis restarts at 8^1, building 9^1, 10^1 and 8^3 (and 10^1 is marked). When 8^3 is reached, the safeguard condition at 8^1 prevents the abstraction that was refined earlier, since it is not possible to find an abstraction of c_{8^1} that both subsumes c_{8^3} and entails $ia < la$. This forces the algorithm to build the descendants 9^2, 10^2 and 8^4 of 8^3. From 8^3, the transition from 8 to 10 is feasible and 10^2 is not marked.

Subsumption of 8^4 by 8^3 (5.7 & 5.8) Once 8^4 is reached, the algorithm attempts to subsume the latter by a previous occurrence of 8. Once again,

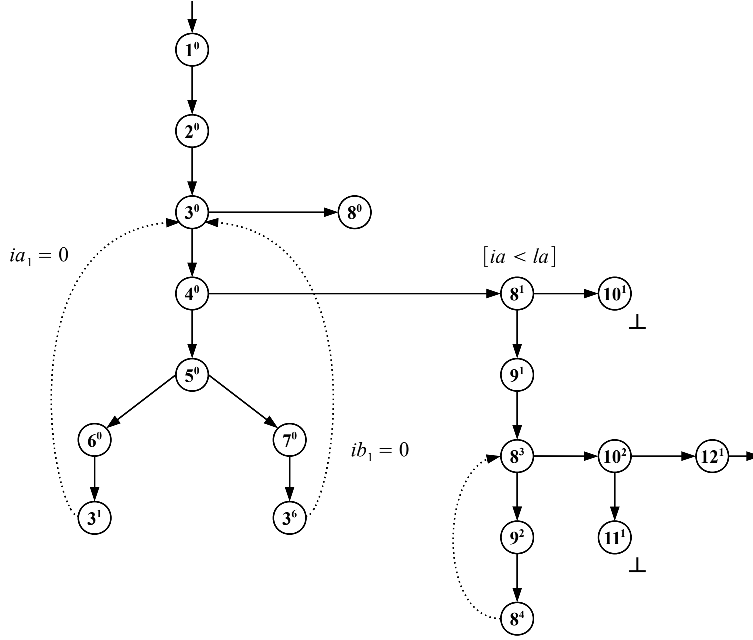


Figure 5.8: A final red vertex is reached without triggering any refinement.

the subsumption by 8^1 fails because of the safeguard condition of the latter. However, the configuration of 8^4 is "naturally" subsumed by c_{8^3}, since both do not require *ia* to be lesser than *la* anymore: the subsumption succeeds without requiring to abstract c_{8^3}.

Subsumption of 10^3 by 10^4 (5.8 & 5.9) The analysis resumes at 10^2 whose successors are built: 11^1 is marked but not 12^1. When the latter is reached, the algorithm starts searching for a too crude abstraction along the shortest path leading to 12^1 but finds none. The vertex on top of *rvs*, 8^0, is processed. Since there are no other occurrences of 8 along the path leading to 8^0, no subsumption can be established and successors of 8^0 are built: 9^3 is marked and we suppose the analysis goes at 10^3. Its descendants are built until 10^4 is processed: the latter is subsumed by 10^3 without requiring any abstraction since no constraint weight over *rv* at both red vertices.

End of the analysis (5.9) At this point, the only element in *rvs* is 12^2: the path leading to 12^2 has not been made feasible by abstraction, and the counterexample guided refinement mechanism does not trigger. The analysis halts, since *rvs* is empty. Since every leaf of the red part is either marked, subsumed or an occurrence of a final location, the resulting LTS, depicted in Figure 5.10, is obtained by removing edges to marked vertices, turning

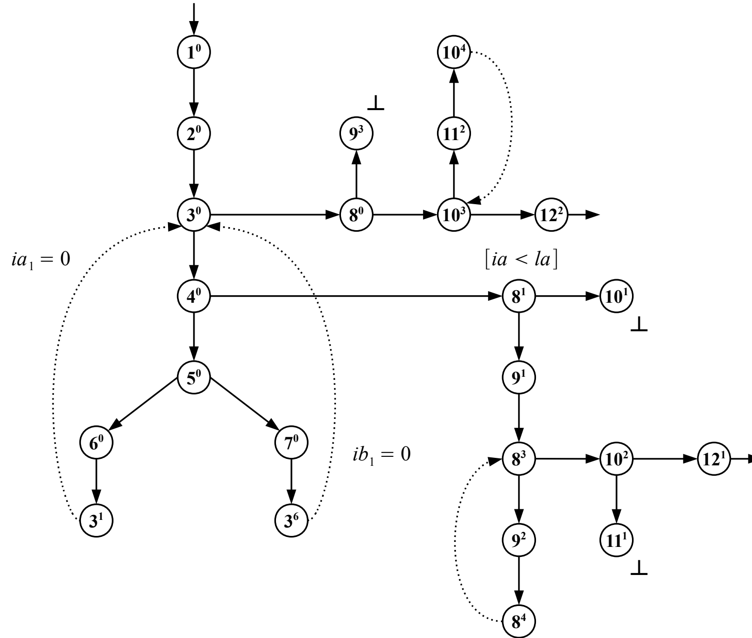


Figure 5.9: The final unfolding: the red part has been completely built.

subsumption links into back edges, renaming vertices of R and labeling its edges according to the transition relation of the black part of RB.

Vertices 14, 8 and 12 are branching nodes, but one of their branches has been removed as infeasible. In the resulting LTS we keep the same sequence of traces as the input LTS, hence the repeated occurrence of $ia \geq la$ on edges $3-14$ and $14-15$ and the unnecessary tests at 8 and 12.

Discussion As one can see in Figure 5.9 or 5.10, a number of infeasible paths of the input LTS shown in Figure 5.3 have been pruned:

- paths that leave the first loop because ia has reached la but enter the second loop,
- paths that leave the first loop because ib has reached lb but enter the third loop
- paths that leave the first loop because ib has reached lb but that do not enter the second loop,
- paths that enter both the second and third loop,

that is, groups of infeasible paths 1, 2, 4 and 7 identified in Section 5.4.1 were detected and pruned. Informally, detecting the infeasibility of paths of groups 5 and 6 would require to remember, when symbolic execution exits

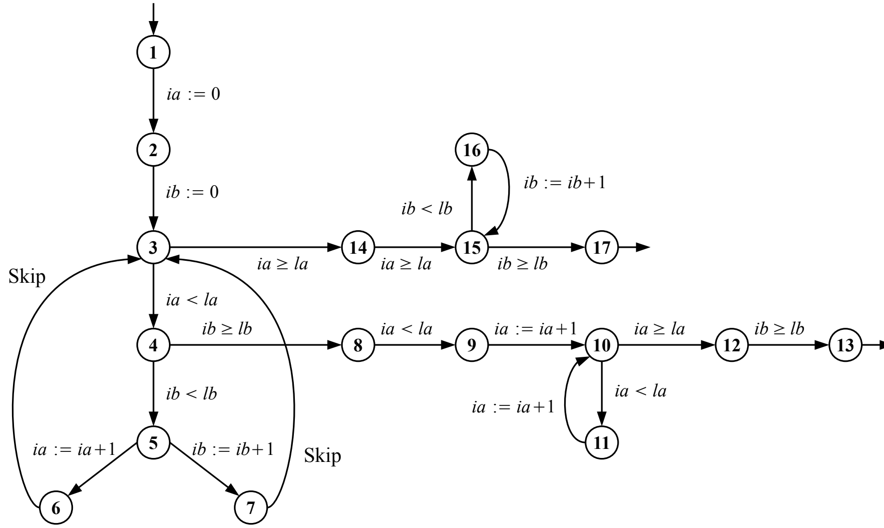


Figure 5.10: The resulting LTS.

the first loop, which index has been incremented last on the current path. This is precisely the information that needs to be abstracted in order to force the subsumptions of 3^1 and 3^2 by 3^0. Since, in the previous example, shortest paths to occurrences of the final location are not elements of groups 5 and 6, the counterexample guided refinement mechanism is unable to reveal the infeasibility of those paths. We will see in 5.4.3 how the look-ahead mechanism allows detecting them.

Abstraction by Store Update

Running our algorithm on the merging sort example with the same parameters than previously except for the abstraction method also yields the LTS shown in Figure 5.10. The analysis follows the exact same steps than in the previous case. This is due to the fact that, in this particular example, although the two abstraction methods actually yield different configurations, those configurations happen to represent the exact same sets of program states in both cases.[3] In this section, we simply illustrate how abstractions are computed and propagated but do not give as many details as previously.

[3]This is generally not the case though, see for instance the short example given at page 116.

Subsumption of 3^1 by 3^0 (Fig. 5.4) When 3^1 is reached, its configuration is

$$c_{3^1} = \big(\{ia \mapsto 2, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\},\\ ia_1 = 0 \wedge ib_1 = 0 \wedge ia_1 < la_0 \wedge ib_1 < lb_0 \wedge ia_2 = ia_1 + 1\big)$$

and the configuration of 3^0 is

$$c_{3^0} = \big(\{ia \mapsto 1, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ia_1 = 0 \wedge ib_1 = 0\big)$$

which does not subsume c_{3^1}.

First, the set of program variables that potentially need to be abstracted at 3^0 in order to force the subsumption of 3^1 is computed. These variables are those that are defined or abstracted along the (shortest) sub-path *sp* leading from 3^0 to 3^1 and that are not already abstracted at 3^0. The only variable defined along *sp* is *ia*, whose symbolic counterpart ia_1 at 3^0 occurs in the path predicate of the latter. Abstracting *ia* at c_{3^0} yields the abstraction

$$a = \big(\{ia \mapsto 2, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\}, ia_1 = 0 \wedge ib_1 = 0\big)$$

which subsumes both c_{3^1} and the abstraction

$$a' = \big(\{ia \mapsto 3, ib \mapsto 1, la \mapsto 0, lb \mapsto 0\},\\ ia_1 = 0 \wedge ib_1 = 0 \wedge ia_2 < la_0 \wedge ib_1 < lb_0 \wedge ia_3 = ia_2 + 1\big)$$

propagated to 3^1.

Subsumption of 3^2 by 3^0 (5.5) When 3^2 is reached, its configuration is

$$c_{3^2} = \big(\{ia \mapsto 2, ib \mapsto 2, la \mapsto 0, lb \mapsto 0\},\\ ia_1 = 0 \wedge ib_1 = 0 \wedge ia_2 < la_0 \wedge ib_1 < lb_0 \wedge ib_2 = ib_1 + 1\big)$$

Once again, the value of *ib* prevents a natural subsumption of 3^2 by 3^1. This case being symmetric to the previous one, it is enough to abstract *ib* at 3^0 to force the subsumption with 3^2.

End of the analysis The next steps of the analysis are similar to those of the previous example. The subsumption of 8^2 by 8^1 requires abstracting *ia* at the latter which causes 10^0 to be unmarked and triggers a refine-and-restart phase when 12^0 is reached. The faulty abstraction is refined and the analysis restarts at 8^1 now labeled with the safeguard condition $ia < la$, which causes the second loop to be unrolled two times instead of one, leading to the subsumption of 8^4 by 8^3. Finally, 8^0 is processed and the analysis ends after detecting the subsumption of 10^3 by 10^4. Once again, the two last subsumptions do not require any abstraction.

5.4.3 Merging Sort with Path Sets Comparisons

Abstraction by Constraint Removal

We still consider that abstractions are performed by removing constraint from path predicates, but we now suppose that the look-ahead mechanism is enabled, and that la (the global parameter) is set to 2. The idea here is to force the algorithm to discover the infeasibility of paths of groups 5 and 6, i.e. those paths that exit the first loop in an inconsistent manner, by comparing feasible path sets starting at each potential subsumee and subsumer up to a depth of 2. Given a red vertex rv to subsume and a potential subsumer rv', the algorithm only checks for subsumption and searches for an abstraction of rv' if the sets of (black) feasible sub-paths of length 2 starting at rv and rv' are equal. In this example, a look-ahead of 2 is enough to discover that, if ia (resp. ib) has been incremented during the last iteration of the first loop, then the execution can only leave the latter through the transition $(3, \text{Assume } ia \geq la, 8)$ (resp. $(4, \text{Assume } ib \geq lb, 8)$).

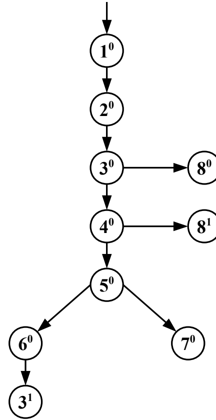


Figure 5.11: Since ib must be lesser than lb at 3^0, the look-ahead mechanism prevents the abstraction at 3^0 needed for a subsumption at 3^1.

First iteration of the first loop: first path (5.11) The analysis starts at the red root 1^0 and follows the first path of the main loop, as in the previous examples, until the second occurrence 3^1 of 3 is reached. The only possible subsumer being 3^0, sets of feasible paths of length 2 starting at 3^0 and 3^1 are built and compared. At 3^0, nothing is known about the value of ia and ib compared to those of la and lb, and the execution can exit the first loop by transitions $(3, \text{Assume } ia \geq la, 8)$ and $(4, \text{Assume } ib \geq lb, 8)$. This is not the case at 3^1, which requires ib to be strictly lesser than lb, making the sub-path $3 \cdot 4 \cdot 8$ infeasible from 3^1 while feasible from 3^0. As

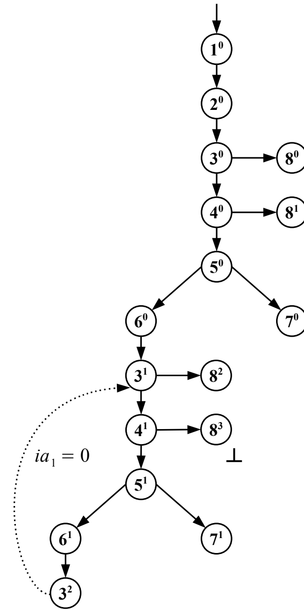


Figure 5.12: The subsumption of 3^2 by 3^1 succeeds.

a result, 3^0 is immediately discarded as a potential subsumer for 3^1, which causes successors of the latter to be built.

Subsumption of 3^2 by 3^1 (Fig. 5.12) We suppose that symbolic execution follows from 3^1 the same path that led from 3^0 to 3^1, until the third occurrence of vertex 3 is reached. The algorithm attempts to subsume 3^2 by one of its ancestor. Let us say that 3^0 is considered first. Since the configuration of 3^2 still requires ib to be lesser than lb, which is not the case at 3^0, the sets of feasible paths starting at 3^0 and 3^2 differ, $3 \cdot 4 \cdot 8$ being feasible from the former but infeasible from the latter. Once again, 3^0 is discarded, and 3^1 is considered. Both configurations of 3^1 and 3^2 requires ib to be lesser than lb, while nothing is known about ia compared to la: sets of feasible sub-paths of length 2 starting at 3^1 and 3^2 are equal. The algorithm now checks for subsumption between their configurations, which fails, since the configuration of 3^1 requires ia to be 1 while the configuration of 3^2 requires it to be 2. This is not a problem: removing $ia_1 = 0$ from the path predicate of c_{3^1} yields an abstraction of the latter that both subsumes c_{3^2} and the abstraction that is propagated to 3^2.

Second iteration of the first loop: second path (Fig. 5.13) The analysis resumes at 7^1, whose only successor 3^3 is built. The algorithm attempts to subsume the latter by a previous occurrence of 3 but fails. At this point, the potential subsumers for 3^3 are 3^0 and 3^1 (3^2 cannot be

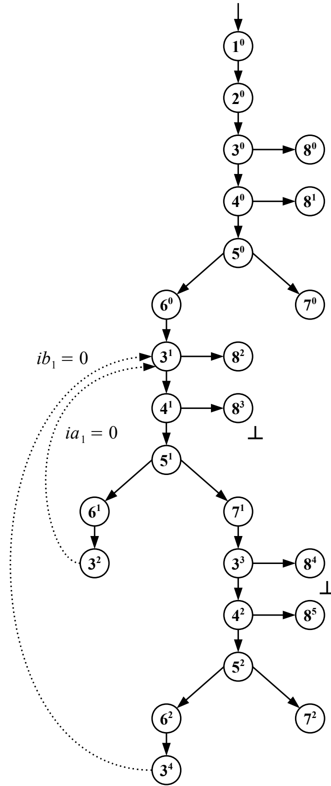


Figure 5.13: No subsumption exists for 3^3 since it is the only occurrence of 3 from which $3 \cdot 8$ is infeasible. The subsumption of 3^4 by 3^1 succeeds.

considered: it is already subsumed and lies on a different path). Indeed, the configuration at 3^3 now requires ia to be lesser than la, which makes the sub-path $3 \cdot 8$ infeasible from there, while it is feasible from both 3^0 and 3^1. Since no subsumer has been found for 3^3, its successors are built, causing the loop to be unfolded a third time.

Subsumption of 3^4 by 3^1 (Fig. 5.13) We suppose again that symbolic execution follows from 3^3 the path along which ia is incremented, until the fifth occurrence of 3 is reached. Potential subsumers for 3^4 are 3^0, 3^1 and 3^3. The first candidate, 3^0 is discarded since it is ia that has been incremented last along the current, causing the configuration of the latter to require that ib be lesser than lb, making $3 \cdot 4 \cdot 8$ infeasible from 3^4. However, feasible paths of length 2 starting at 3^4 are exactly those starting at 3^1: the algorithm checks that c_{3^4} is subsumed by c_{3^1}, but fails. Indeed, the configuration at 3^1 requires that ib is 0 while c_{3^4} requires it to be 1, since ib has been abstracted between 3^0 and 3^3. Removing $ib_1 = 0$ from the path predicate of 3^0 once again rules the problem, since it yields an abstraction that subsumes c_{3^4} and

the abstraction propagated to 3^4 and that can safely be propagated to 3^2.

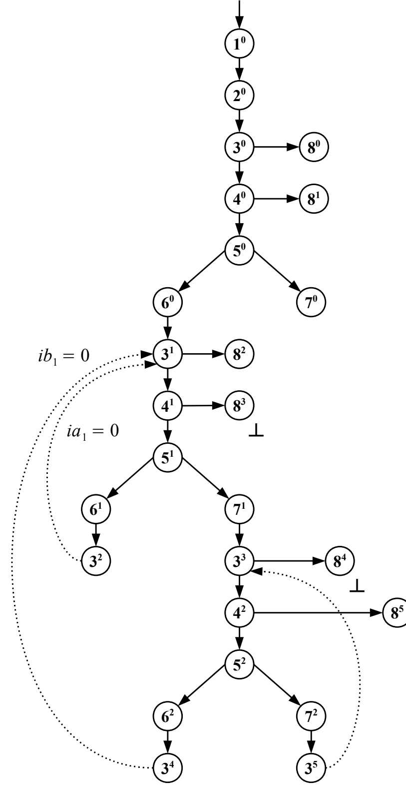


Figure 5.14: Sets of feasible sub-paths starting at 3^3 and 3^5 are equal: the subsumption of the second by the first succeeds.

Subsumption of 3^5 by 3^3 (Fig. 5.14) The next red vertices on the stack, 7^2, is processed and the analysis reaches its only successor 3^5. Potential subsumers for 3^5 are 3^0, 3^1 and 3^3, the two other occurrences of 3 lying on different symbolic paths. Since ib has been incremented last on the path leading to 3^5, the set of feasible paths of length 2 starting at the latter differ from those of 3^0 and 3^1, but is equal to the one starting at 3^3. Since both abstractions performed at 3^1 have been propagated to 3^3, the first check for subsumption between c_{3^5} and c_{3^3} succeeds and the subsumption is established without requiring any abstraction of the latter.

Subsumption of 8^7 by 8^6 (Fig. 5.15) The execution resumes at 8^5, the only occurrence of 8 along the current symbolic path. Its successors 9^0 and 10^0 are built, the latter being marked since its configuration requires ia to be both lesser and greater or equal to la. Descendants of 8^5 are built until 8^6 is reached, the algorithm trying to subsume the former by the latter. Here,

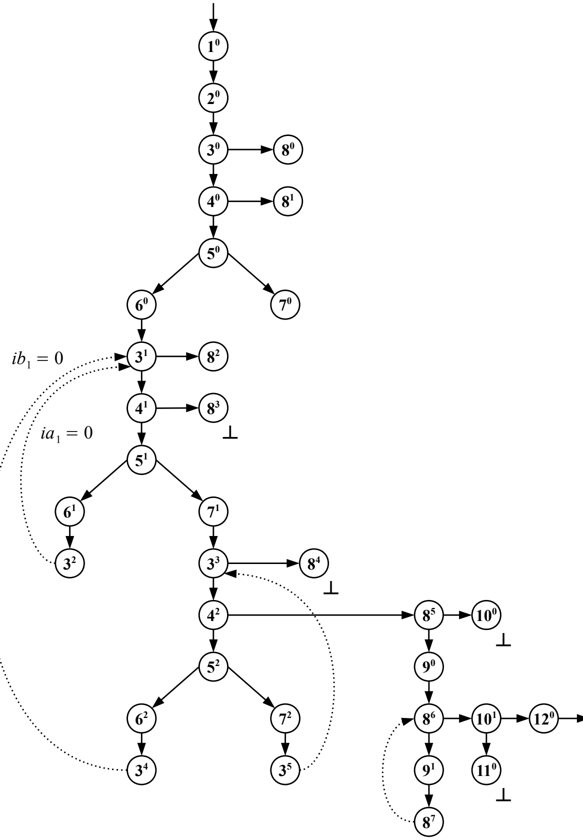


Figure 5.15: The look-ahead mechanism prevents the abstraction of 8^5. The subsumption between 8^7 and 8^6 avoids triggering a refinement step.

the look-ahead mechanism detects that the execution can safely exit the second loop at 8^6 while this is impossible at 8^5, causing the subsumption attempt to fail. Descendants of 8^6 are built, until 8^6 is reached. Once again, the attempt to subsume the latter by 8^5 fails for the same reasons, but succeeds at with 8^6, since c_{8^6} and c_{8^7} do not impose $ia < la$ anymore. Once subsumption has been established, the first final occurrence of a final location is reached. Since the feasibility of the path leading to 12^0 is not due to some previous abstraction, the counterexample guided refinement mechanism does not trigger.

End of the analysis (Fig. 5.16) We do not detail the rest of the analysis as much: this would be unnecessary long and all special cases that occur are symmetric to those presented earlier. For example, when the analysis resumes at 8^2, the algorithm would try to subsume its descendant 10^3 by 10^2, but the look-ahead mechanism acts as for 8^6 and 8^5 and prevents the

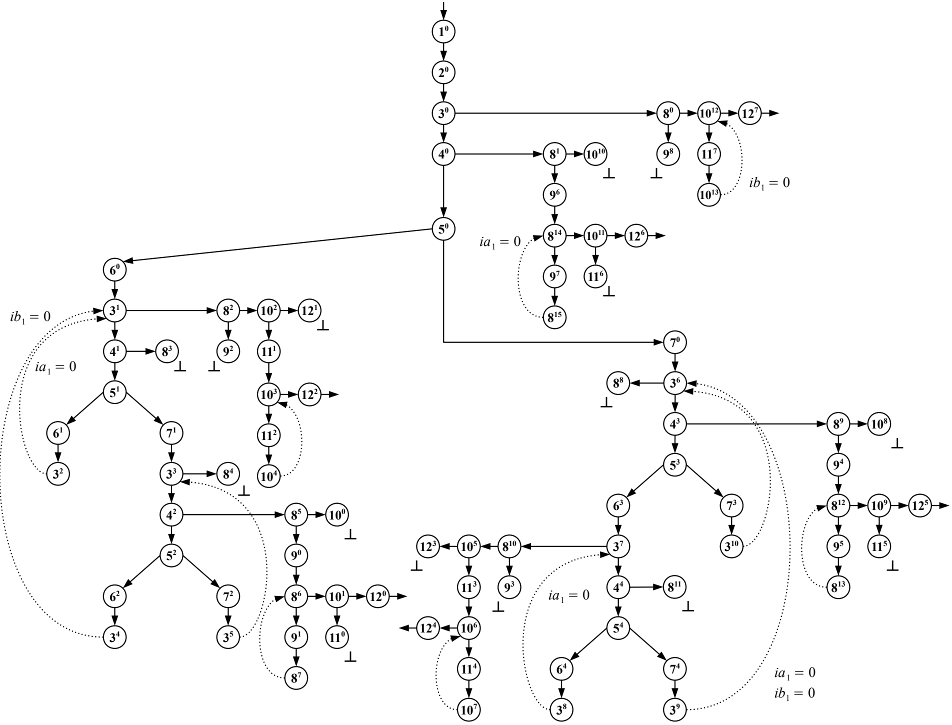


Figure 5.16: The complete red part at the end of the analysis. No infeasible paths remain.

first subsumption possible, which causes the third loop to be unfolded a second time. This also happens at each first unmarked occurrence of 8 (in this case, the second loop is unfolded twice) or 10 along the other symbolic paths, 8^0 excepted, since the execution can enter the third loop or not from there. The more interesting case comes from the execution of the second path at the first iteration of the main loop: the latter is unfolded at least two more times from 3^6 in order to remember, at each occurrence of loop headers, which index was incremented last, similarly to what happened at 3^1, the situation being symmetric at those two red vertices.

Discussion We do not show the LTS obtained from the complete red part shown in Figure 5.16. It is made of 61 vertices, 72 transitions, and does not contain infeasible paths anymore. Unsurprisingly, infeasible paths that were eliminated in the previous section, when the look-ahead mechanism was disabled, were also detected here with the global parameter *la* set to 2. Most importantly, this mechanism allowed to detect and prune paths whose infeasibility is caused by complex dependencies between iterations of one or several loops of the program, as illustrated by this example, and that cannot be detected by our counterexample guided refinement mechanism in its

current form. Also, we insist on the fact that no particular precondition was used: we did not bound the length of the input arrays, the length of paths or the level of loop unfoldings. The set of paths of the LTS in Figure 5.16 is exactly the set of feasible paths of the input LTS.

Finally, we observe that several regions of the red part shown in Figure 5.16 are strictly identical, up to a renaming of red vertices. This is due to the facts that *(i)* we chose, at the beginning of this section, to only consider subsumption links involving red vertices that occur along the same symbolic paths and *(ii)* that some patterns of execution of the bodies of the different loops are repeated along the different symbolic paths exhibited in the red part. In other words, although the red part of Figure 5.16 only contain feasible paths, its set of paths is not minimal w.r.t. the set of feasible paths of the input LTS. In Chapter 6, we discuss a number of experiments that we have done in order to over-approximate the sets of feasible paths more closely, by trying to merge those similar regions.

Abstraction by Store Update

We now describe how the algorithm behaves when abstractions are performed by updating stores, with the look-ahead mechanism enabled and the global parameter la set to 2. In this case, an infinite chain of refine-and-restart phases occurs, preventing the analysis to halt unless a length limit for symbolic paths has been set through the global parameter mrl.

At first, the analysis goes on as in the previous case (see Figure 5.17): the subsumption of 3^1 by 3^0 is refused, but 3^2 then gets subsumed by 3^1 by abstracting ia, and, when 3^3 is reached at the end of the second path in the first iteration of the main loop, the latter is unfolded once again, yielding the subsumption of 3^4 by 3^1, which requires to abstract ib at 3^1. After this subsumption has been established, the configuration at 3^1 is

$$\begin{aligned} c_{3^1} = \big(&\{ia \mapsto 3, ib \mapsto 2, la \mapsto 0, lb \mapsto 0\}, \\ &ia_1 = 0 \wedge ib_1 = 0 \wedge ia_1 < la_0 \wedge ib_1 < lb_0 \wedge ia_2 = ia_1 + 1\big) \end{aligned}$$

which does not require ia or ib to be lesser than la and lb, respectively. When this abstraction is propagated from 3^1, the fact that the constraint $ib < lb$ was forgotten causes the red vertex 8^3 to be unmarked: from there, the second occurrence 12^1 of the final black vertex 12 is reached, which triggers refine-and-restart mechanism for the first time.

The subsumption of 3^4 by 3^1 is refined, and the analysis restarts from the latter (see Figure 5.18). The subsumption of 3^6 (which previously was 3^2) by 3^1 is established: the refine-and-restart step was not due to the abstraction of ia and the required abstraction is not prevented. However, when the analysis reaches 3^7 (which previously was 3^3), the situation at the latter is identical, although symmetrical, to the situation at 3^1 before the faulty

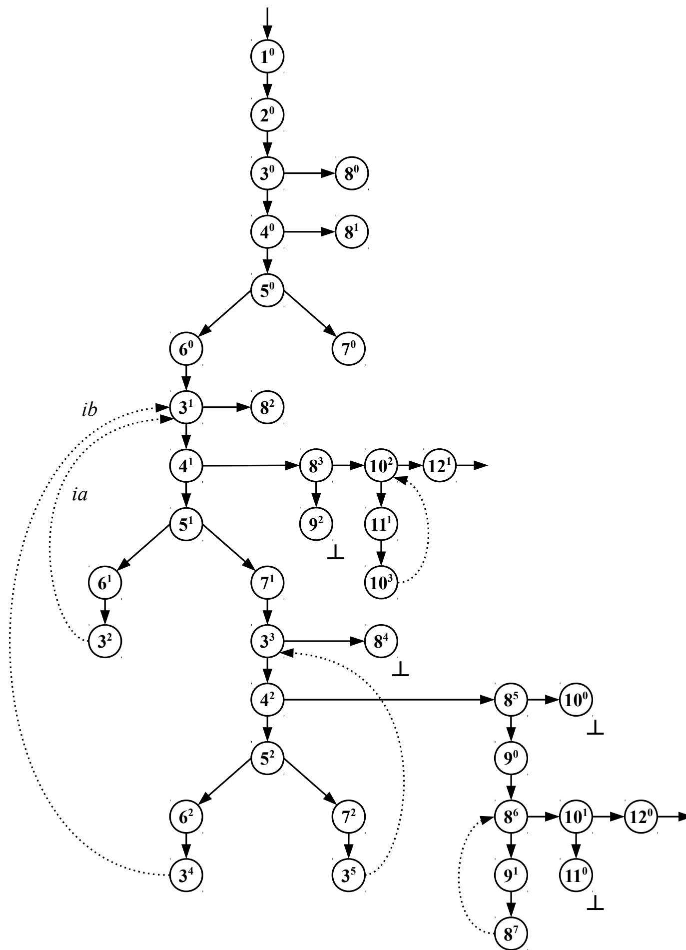


Figure 5.17: First partial red part: the subsumption of 3^4 by 3^1 causes 8^3 to be unmarked, triggering a refine-and-restart step at 12^1.

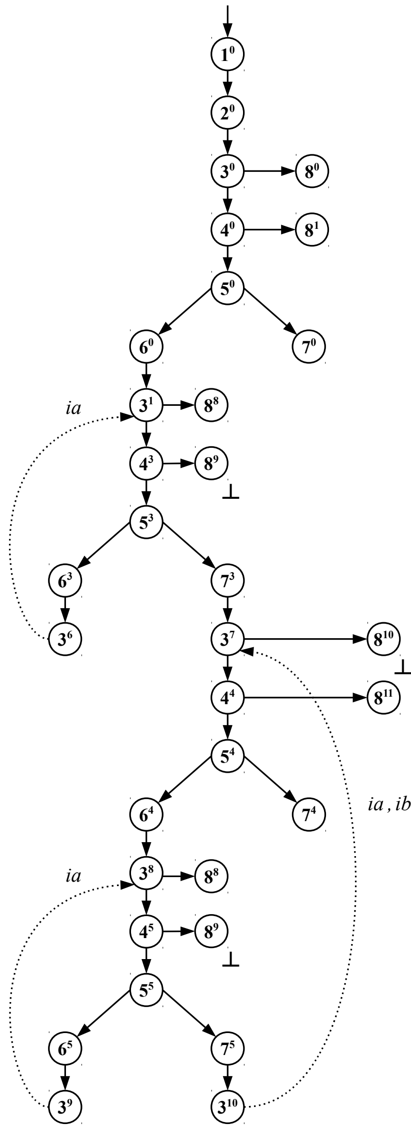


Figure 5.18: Second partial red part. The situation is similar and symmetrical: the subsumption of 3^{10} by 3^7 causes 8^{10} to be unmarked, which also triggers refine-and-restart phase.

abstraction. A few steps later, the algorithm establish the subsumption of 3^{10} by 3^7, which requires abstracting both *ia* and *ib*. This in turn will cause the red vertex 8^{11} to be unmarked (although, this time it will be reached from an occurrence of 3, since the situation is symmetric) which will cause a refine-and-restart phase to trigger. This process goes on infinitely.

In such cases, there are two options for forcing the algorithm to halt. The first consists in bounding the maximal limit of symbolic paths through *mrl*. This usually yields an incomplete red part decorated with subsumption links that might over-approximate the set of feasible paths of length lesser or equal to *mrl* very closely. In the case of merging sort, all infeasible paths of length lesser or equal to *mrl* are detected and pruned. However, since the red part is incomplete, the resulting LTS will need to be completed with transitions coming from the input LTS: the accuracy of the result significantly drops when considering paths whose length is greater than *mrl*. Since our ultimate goal is drawing, at random, paths of a given maximal length in LTS, and since such LTS are usually very accurate for paths of length lesser or equal to *mrl*, bounding the length of symbolic paths while keeping the counterexample guided mechanism enabled might yield the more accurate results if we know in advance we are not in interested in drawing paths longer than *mrl*. For the merging sort example, with *mrl* set to 30, we obtain a LTS with 142 vertices and 183 transitions, 14 of them coming from the black part.

The second option simply consists in disabling the counterexample guided refinement mechanism through the boolean flag *restart*.[4] Naturally, this causes some infeasible paths not to be detected, as illustrated by the previous example, but this also allows to build a complete red part. Informally, this yields LTS that might be less accurate than those obtained by simply bounding the height of the red part when considering paths of length lesser or equal *mrl*, but are then a lot more precise on the long run, when the length of paths exceeds *mrl*. Thus, we might prefer not to bound the length of symbolic paths but disable the counterexample guided mechanism if we do not know in advance the maximal length of paths we are going to draw in the resulting LTS. In the case merging sort, disabling the counterexample guided refinement mechanism, but keeping the same parameters than before yields a LTS with 91 vertices and 108 transitions, but that still contain some infeasible paths.

The next chapter includes all numbers related to this example, for various combinations of values for the parameters, and reports on other experiments.

[4]Disabling the the counterexample guided refinement mechanism does not make the length limit *mrl* useless: non-termination might come from the look-ahead mechanism preventing subsumptions; although not in the case of merging sort.

5.5 Summary

In this chapter, we have described our algorithm in detail. Its principle is, given an initial configuration and an input LTS that will serve as the black part of an intermediate red-black graph, to build the red part of the latter according to a DFS traversal of its black part. It then returns a LTS whose set of paths is exactly the set of red-black paths of the intermediate red-black graph. We described each of the three actions that can be performed at each red vertex, i.e. refine a previous abstraction and restart the analysis from there, subsume it by a previous occurrence of the black vertex it represents if the latter is a loop header, or build its successors using symbolic execution.

We also described the various heuristic aspects our algorithm relies on in order to combine and perform those three actions, for example: how candidates for subsumption can be chosen, how abstractions can be computed, combined and propagated, how faulty abstractions are refined and how safeguard conditions are computed.

Finally, we described in details how our algorithm behaves on the example of merging sort. As revealed by this example, the results greatly vary when the algorithm is seeded with different values of its parameters.

The goal of this last section — and of this chapter more generally — was more to illustrate how the different features of our algorithm interact than to present quantitative results. This is the object of the next chapter. As shown during this chapter, the three possible actions interact in a rather complex and intricate manner. This is why we felt a machine-aided formalization was needed to prove the key properties of this approach and why we chose to model it as a set of five basic operators, assuming all possible choices were done in advance, rather than try to model the whole algorithm itself. This is also what motivates introducing red-black graphs and related notions as well as our formalization before the algorithm itself. Although the algorithm was not proven itself, it was designed to be as close as possible to the model we used in our formalization, which gives strong confidence in its correctness.

In this chapter, we have mainly proposed simple solutions to the heuristic aspects of the problem: these are the heuristics we actually use in our implementation of this approach and with which we obtained the results that are presented in the next chapter. Nonetheless, those simple heuristics provide yet fairly good results on most examples, as we will see in the next chapter. Although we are conscious that the effectiveness of the overall approach might be improved with more elaborated heuristics — and we mentioned some perspective for the counterexample guided refinement mechanism in that purpose, the goal of this chapter was more to present the details of our approach rather than to discuss its possible improvements. This will also be done in the second part of the next chapter, after experimental results have been presented.

Chapter 6

Experiments and Discussions

In this chapter, we first present experimental results obtained with our implementation of the algorithm described in the previous chapter. The goal of the experiments reported in this chapter is to assess the infeasible path detection power of our approach on various programs and to compare our different heuristics for handling subsumptions and abstractions.

The second section of this chapter is dedicated to discussing the limitations of our approach in its current state and some possible improvements, at the light of the experimental results.

6.1 Experimental Results

For each of the programs that follow, we first give its pseudocode and LTS, briefly describe its global behavior and detail its set of infeasible paths. Then, we build various new LTS using different values of the parameters of our tool: the way abstractions are computed, allowing refinements or not, and the depth of the look-ahead. Finally, for each LTS obtained, we compare the number of *complete* paths, i.e. paths from the initial location to any final location, of a given maximal length l with the corresponding number of *complete* feasible paths, and interpret the results. In the following, we are only interested in complete paths, and will refer to them simply as paths. Since our approach yields classical symbolic execution trees in absence of loops, we do not give results for such cases: all the following examples contain at least one loop.

We insist on the fact that our purpose in this chapter is not to generate actual test cases nor to draw paths at random, but to compare the respective numbers of paths and feasible paths in the input LTS and in the LTS produced by our approach. In a complete testing environment, as described in [20], test data would be generated by symbolic execution and constraint solving methods along paths drawn from the LTS produced with our prototype, the drawing being performed according to some coverage criterion

using the RUKIA library [45].

In the experiments reported here, paths were counted using the Rukia library. Rukia implements a number of algorithms that allow to draw paths at random in huge graphs. It takes as inputs a graph g (written in `.dot` format), a set of vertices of interest V (in our case the final vertices of the input graph), a number of paths n to draw and a length l, and returns *(i)* the number $P_{=l}$ of paths of length l going from the initial vertex of g to any element of V and *(ii)* n paths of length l, drawn uniformly among those $P_{=l}$ possible paths. Here we use Rukia as an efficient counting device as well as for its capability to uniformly draw long paths in large graphs. Since we are interested in paths of length at most l and since Rukia only produces paths of length exactly l, whenever we send a graph g to Rukia, we add an edge going from its root to itself: the number $P_{=l}$ of paths of length l in this extended graph is exactly the number $P_{\leq l}$ of paths of length at most l in g.[1]

The experiments reported in this section were performed using the following protocol, given an input LTS $\mathcal{S}$:

1. we count the number of paths (feasible or not) of length at most l in $\mathcal{S}$ using Rukia, for $l = 30$, 50 and 100,

2. we count the number of feasible paths of length at most l in $\mathcal{S}$, for $l = 30$ and 50. To do so, we first build the classical symbolic execution tree (SET) of height 50 obtained from $\mathcal{S}$ using our implementation — we simply disable subsumption checks and set *mrl* to 50. Then, we count the number of paths of length at most 30 and 50 in this SET: unless the solver is unable to prove the infeasibility of some of these paths (and this never happened in these experiments), they correspond to the feasible paths of length at most 30 and 50 of $\mathcal{S}$. It is not needed to build the SET of height 30 of $\mathcal{S}$, since it is a sub-tree of the SET of height 50: we count paths of length at most 30 in the latter.

   For paths of length at most 100, we have proceeded differently: due to their size, it was not possible to build the SET of height 100 of those example programs and to count this way the number of feasible paths of length at most 100 in their LTS. When this number is included in our report, it was "deduced" by checking by manual inspection that on this particular example our resulting LTS contains no infeasible path: the number is simply the total number of paths, as reported by Rukia. Such numbers have a trailing * mark in the tables. When the method does not apply because some infeasible paths remain in our LTS, we do not include the number in the report.

[1] This slight modification of the graph sent to Rukia is equivalent to the modification introduced in Chapter 2, since we consider LTS whose transitions never lead to their initial locations.

3. we produce a LTS $\mathcal{S}'$ using our implementation, and count its number of paths of length at most 30, 50 and 100, and compare them to the numbers of paths and feasible paths obtained for the input LTS $\mathcal{S}$ at steps 1 and 2,

4. finally we count the number of feasible paths of $\mathcal{S}'$ in order to check that no feasible paths were lost during the analysis. To do so, we proceed differently than at step 2. For $l = 30$ and 50, we use Rukia not only for counting $P_{\leq l}$, the number of paths of length at most l in $\mathcal{S}'$, but also for enumerating all these paths, before counting the feasible ones among those using our implementation. Since our approach preserves feasible paths, those numbers must be the same as those obtained at step 2. To enumerate all paths in $\mathcal{S}'$ we repeatedly ask Rukia to draw $P_{\leq l}$ paths, adding them (without duplicates) in a global collection. Note that, when asked to draw n paths, Rukia usually does not draw n distinct paths, hence we have to repeat the drawing until all the $P_{\leq l}$ distinct paths have been collected. Surprisingly — or not, considering *(i)* the combinatorial explosion of the number paths when building symbolic execution trees of large heights and *(ii)* the efficiency of Rukia and its uniform drawing capability, in all our examples counting in this way the number of feasible paths of length at most l in S' was effective and usually done a lot faster — when $P_{\leq l}$ is not too large — than by building the classical SET, then counting its paths.

   All results below were checked in this way. As expected, no loss of feasible paths was ever revealed with the current version of our implementation.

Since our goal is to count paths and feasible paths in the input and resulting LTS, the choice of 30, 50 and 100 for l may seem arbitrary, excepted for the facts that *(i)* we think that they describe fairly well the evolution of the ratios of feasible paths over paths of a given maximal length in the LTS produced by our approach and *(ii)* tracking the number of paths of lengths greater than 100 would not be useful in our case, since we are not able to build SET and count the feasible paths for such values of l: we would not be able to check that we did not loose some feasible paths. We choose to stick to those values through each example to give a better idea of the evolution of sets of paths and feasible paths according to l. We recall that l is not (a bound on) the length of paths that would be drawn for test case generation; such a bound should be chosen by testers depending on the structure of the LTS of the program under analysis.

Tables giving the experimental results are organized as follows:

- the first column indicates which LTS, $\mathcal{S}$ or $\mathcal{S}'$, is considered,
- the second column, noted a, shows which method of abstraction was

used: with $a = 1$ abstractions were performed by removing constraints, and by updating stores with $a = 2$,

- a mark in the third column, noted r, indicates that refinements were allowed,
- the fourth column, noted $|L|$, gives the number of vertices of the considered LTS,
- the fifth column, noted $|\Delta|$, gives its number of transitions,
- the sixth column, noted t, gives the time, in seconds, needed to build the corresponding LTS. All experiments were performed on a Linux machine equipped with an Intel Core i5-3320M processor at 2.6 GHz and 8 GB RAM. In our current implementation, we only used one constraint-solver: the Z3 theorem prover (version 4.01),
- finally, the remaining columns give the number of paths (P) and feasible paths FP for each value of l.

We recall that, although it is not our goal in this chapter, the LTS we build using our approach will ultimately be used to draw paths in a white-box testing context, i.e. test data will be produced from paths drawn in our resulting LTS, using constraint solving techniques. Therefore, we do not allow here, unlike in the previous chapter, to modify the structure of the underlying graph of our input LTS as we did for the merging sort example in 5.4. However, we still consider that it is possible to abstract the labels of some transitions when those labels have no influence on the feasibility of paths or are not supported by our input language. When such an abstraction is performed, one has to recall that paths drawn in resulting LTS might not have the same trace than their equivalent in the original input LTS, and that labels that were abstracted must be restored before test data generation, either in the resulting LTS or, on the fly, along the paths that will be drawn.

6.1.1 Greatest Common Divisor

We first consider a program implementing Euclid's algorithm for computing the greatest common divisor of two integers a and b. The pseudocode of this program is given in Figure 6.1, its LTS is depicted in Figure 6.2.
The LTS of Figure 6.2 contains three groups of infeasible paths:

- paths that enter the external loop, but that do not enter any of the internal ones,
- paths that traverse the first internal loop, do not enter the second one and finally enter the external loop again,
- paths that exit the external loop, then enter it again and do not enter the first internal loop.

```
Function gcd(int x, int y)
1   let a = x;
2   let b = y;
3   while a ≠ b do
4       while a > b do
5           a ← a − b;
6       while b > a do
7           b ← b − a;
8   return a;
```

Figure 6.1: The greatest common divisor program.

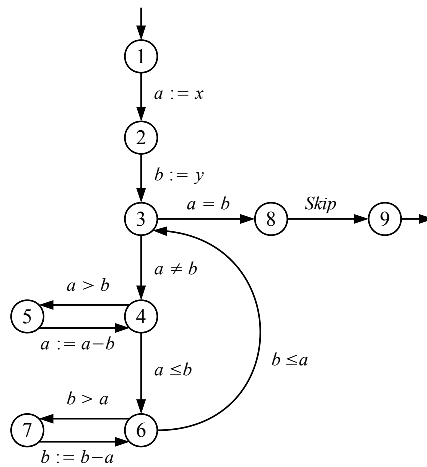


Figure 6.2: The LTS of the greatest common divisor program.

These dependencies between the three loops of the program are simple: the set of feasible paths of *gcd* is a regular language. The number of paths and feasible paths for *gcd* are given in Table 6.1. Note that this program is supposed to be called with positive values for its arguments x and y, otherwise it would not terminate. Experiments described below were done without considering the precondition $x \geq 0 \wedge y \geq 0$, but we remark that results shown in Table 6.1 would be exactly the same if we were to do so, since this precondition has nothing to do with the infeasibility of paths (guards of *gcd* only compare the value of a to b), but only with the termination of the *gcd* program.

Both methods of abstraction yield the same results on this example. If *la* is set to 0 or 1, the algorithm does not detect infeasible paths: in both cases it yields the input LTS. If *la* is set to 2 all infeasible paths are detected, with both abstraction methods. The rightmost column of Table 6.1 gives the number of feasible paths of length at most 100 in $\mathcal{S}$ and in each $\mathcal{S}'$: as said previously, we did not compute this number by building the SET of

Table 6.1: Paths and feasible paths in *gcd*.

<table>
<tr><th rowspan="2"></th><th rowspan="2">a</th><th rowspan="2">r</th><th rowspan="2">la</th><th rowspan="2">$|L|$</th><th rowspan="2">$|\Delta|$</th><th rowspan="2">t</th><th colspan="2">$l = 30$</th><th colspan="2">$l = 50$</th><th colspan="2">$l = 100$</th></tr>
<tr><th>P</th><th>FP</th><th>P</th><th>FP</th><th>P</th><th>FP</th></tr>
<tr><td>$\mathcal{S}$</td><td></td><td></td><td></td><td>9</td><td>11</td><td></td><td>15 478</td><td rowspan="3">792</td><td>4.5×10^{7}</td><td rowspan="3">143 179</td><td>2×10^{16}</td><td rowspan="3">5.9×10^{10} *</td></tr>
<tr><td rowspan="2">$\mathcal{S}'$</td><td>1</td><td>✓</td><td rowspan="2">2</td><td rowspan="2">37</td><td rowspan="2">42</td><td>8.8</td><td rowspan="2">792</td><td rowspan="2">143 179</td><td rowspan="2">5.9×10^{10}</td></tr>
<tr><td>2</td><td>✓</td><td>11.3</td></tr>
</table>

height 100 of *gcd*, but deduced it from the $\mathcal{S}'$ described at line 2 and 3.

Although the *gcd* program is fairly small and the reason of infeasibility of its paths are simple, the *gcd* program illustrates pretty well the need for infeasible path detection techniques in the context of white-box testing: drawing a path uniformly in the input LTS, the probability of it being feasible would only approximately be 0.05, 0.003 and 0.000003, for $l = 30$, 50 and 100, respectively.

6.1.2 Merging Sort

Table 6.2 shows the results for the merging sort example. As explained in the introduction, our values of l are arbitrary and used only to provide measures for infeasible path elimination. To give an idea, for this example, of the kind of test cases that would be considered when also fixing at l the length of a path corresponding to an actual test case, complete paths of length up to 30, 50 and 100 respectively represent executions of merging sort with a bound of 7, 14, and 31 on the size of the resulting array T (implicitly limiting the sizes of the two input arrays that produce it). The numbers of paths and the ratio of feasible paths among them are already impressive and such bounds on the sizes of the input arrays allow to generate test data corresponding to very different cases.

The first line gives the numbers of paths and feasible paths, for each value of l, for the simplified LTS $\mathcal{S}$ shown in Figure 6.4. For *merging sort*, the respective ratios of feasible paths on paths in the input LTS for $l = 30$, 50 and 100 are approximately 0.138, 0.115 and 0.112.

The remaining lines give those numbers for the resulting LTS $\mathcal{S}'$ obtained with our algorithm, for different values of its control parameters.

The second and third lines show the results for the LTS $\mathcal{S}'$ obtained while enabling counterexample guided refinements, but disabling the look-ahead mechanism. This corresponds to the examples described in 5.4.2. In both cases, the two methods of abstraction yield the same LTS, which is depicted in Figure 5.10.

The fourth line shows the number of paths for the LTS obtained by abstracting configurations by constraint removal, allowing refinements and setting the look-ahead depth to 2, which yields the LTS shown in Figure 5.16. As seen in 5.4.3, no infeasible paths remain, which allows to deduce the

```
    Function merge(int[] a, int[] b, int la, int lb, int[] T)
1     let ia = 0;
2     let ib = 0;
3     while ia < la do
4       if ib < lb then
5         if a[ia] < b[ib] then
6           T[ia + ib] ← a[ia];
7           ia ← ia + 1;
          else
8           T[ia + ib] ← b[ib];
9           ib ← ib + 1;
        else
10        break;
11    while ia < la do
12      T[ia + ib] ← a[ia];
13      ia ← ia + 1;
14    while ib < lb do
15      T[ia + ib] ← b[ib];
16      ib ← ib + 1;
```

Figure 6.3: The merging sort program.

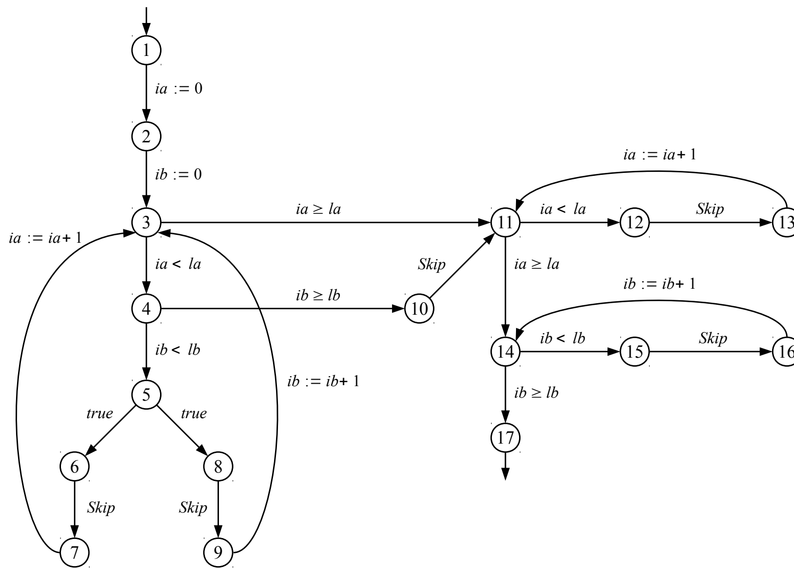


Figure 6.4: The merging sort simplified LTS.

Table 6.2: Paths (P) and feasible paths (FP) in *merging sort*.

	a	r	la	$\|L\|$	$\|\Delta\|$	t	$l = 30$		$l = 50$		$l = 100$	
							P	FP	P	FP	P	FP
$\mathcal{S}$				17	21		593	82	11 728	1 351	12 389 030	1 385 616 *
$\mathcal{S}'$	1	✓	0	23	26	3	210		3 694		3 819 743	
	2	✓				2.9						
	1	✓	2	85	96	15.7	82		1 351		1 385 616	
	2	✓		170	178/19	117.7	82		2 728		11 593 094	
	2			127	144	9.3	131		2 031		1 980 403	

number of feasible paths for $l = 100$.

The fifth line gives the results obtained with the second method of abstraction, allowing refinements and setting mrl to 30 to allow termination. In this setting, a chain of refinements occurs, as described in 5.4.3. The red part is no longer complete (the slash-separated numbers in column Δ are the numbers of transitions coming from the red and black parts, respectively) and $\mathcal{S}'$ is obtained by connecting the red vertices that were not expanded to the black part. A path of length at most 30 is entirely in the red part, but paths of length more than 30 can end in the black part rather than in the red one. As a result, the accuracy of $\mathcal{S}'$ w.r.t. the set of feasible paths of $\mathcal{S}$ decreases dramatically as l grows: the respective ratios of feasible paths on paths are 1, 0.495 and 0.119 for $l = 30$, 50 and 100. We observe however that the red part is not totally devoid of subsumption links, and that paths of length more than 30 might also end in the red part, although this is less and less probable as their length grows. The fact that $\mathcal{S}'$ contains significantly more vertices and transitions than in the other cases is due to the fact that there is a majority of paths of length 30 in the red part.

The sixth line shows the results for the LTS obtained with the second method of abstraction, with refinements disabled and la set to 2. Since refinements have been disabled, the algorithm is able to produce a complete red part, but this comes at the cost of allowing subsumptions that are known to introduce infeasible paths in the resulting LTS. However, we remark that, although the resulting LTS is less accurate than in the previous case for paths of length at most 30, the ratio of feasible paths on paths being 0.626, it is a lot more accurate than in the previous case as l grows, the ratios for $l = 50$ and $l = 100$ being 0.665 and 0.699, respectively.

The resulting LTS corresponding to the last line is less precise than the one computed for the fourth line, but also contains more vertices and edges than the latter, which might come as a surprise. This is due to the fact that, in the case of merging sort, the second method of abstraction, unlike the first, sometimes forgets the fact that $ia < la$ at some occurrences of location 3. Suppose now that, from such an occurrence, the execution does

another iteration of the main loop and increases ib along that path. When the next occurrence of 3 is reached, we have that $ia < la$ and the look-ahead prevents the subsumption since sets of feasible paths differ from those two occurrences. This causes the main loop to be unfolded again, which was not necessary when removing constraints, and leads to a larger S'.

Results obtained with the *merging sort* example are promising since all infeasible paths were detected in the best case (fourth line of Table 6.2) and that the set of feasible paths of $\mathcal{S}$ is fairly closely over-approximated in the second best case (last line of Table 6.2). Moreover, we did not use any kind of precondition to handle merging sort, be it a bound on the length of symbolic paths (excepted for the experiment described at the fifth line of Table 6.2) or on the length of the input or output arrays. To our knowledge, *merging sort* is systematically handled with such bounds in related works.

6.1.3 Substring

We consider the *substring* program, which takes two arrays of characters s_1 and s_2 and their respective lengths l_1 and l_2, and returns 1 (i.e. *true*) if and only if the string s_2 occurs in s_1. Its pseudocode and its simplified LTS are given in Figures 6.5 and 6.6, respectively.

In the latter, the two return statements have been turned into Skip labels. Also, since we do not support array expressions at the moment, the conditions of the guards of transitions going from location 5 have been replaced by *true*. This does not impact results provided by our algorithm: those guards have no influence on the feasibility of paths of the LTS of *substring*: truth values of the expression $s_2[j] = s_1[i + j]$ at a given iteration of the external loop have no impact on the truth values of this same expression in the following iterations of the external loop.

```
   Function susbtring(char[] s1, int l1, char[] s2, int l2)
1    let i = 0;
2    while i ≤ l1 − l2 do
3      let j = 0;
4      while j ≠ l2 do
5        if s2[j] = s1[i + j] then
6          j ← j + 1;
         else
7          break;
8      if j = l2 then
9        return 1;
10     i ← i + 1;
11   return 0;
```

Figure 6.5: The substring program.

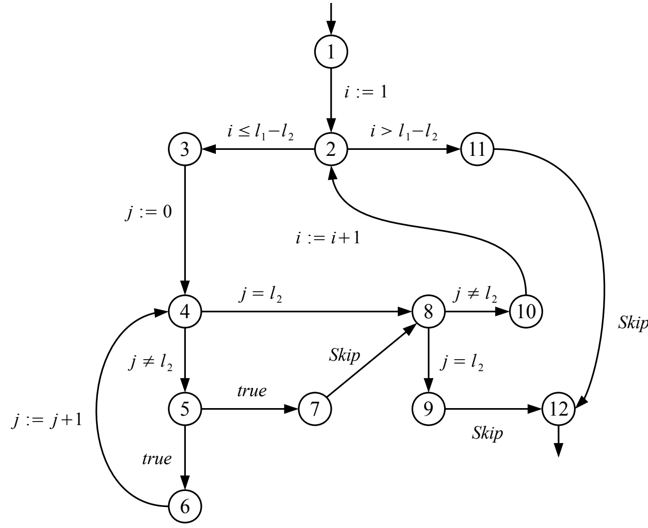


Figure 6.6: The substring LTS.

The LTS depicted in Figure 6.6 contains three groups of infeasible paths. Paths that exit the internal loop with j equals to l_2 and exit the external loop through the transition leading from 8 to 10 are infeasible. Symmetrically, paths that exit the internal loop with j different than l_2 but exit the external loop through the transition leading from 8 to 9 are also infeasible. The infeasibility of paths of the third group is caused by strong dependencies between the external and the internal loop. For example, let s be a strict prefix of s_2 of length l_s found to be a substring of s_1. Then the length of s_2 is greater than l_s and no later iteration of the outer loop can return 1 without doing at least $l_s + 1$ comparisons. As a result, the set of feasible paths of *substring* is not a regular language and, as such, we do not expect to represent it by a graph.

Table 6.3: Paths and feasible paths in *substring*.

	a	r	la	$\|L\|$	$\|\Delta\|$	t	$l = 30$		$l = 50$		$l = 100$
							P	FP	P	FP	P
$\mathcal{S}$				12	15		789		85 598		1×10^{10}
$\mathcal{S}'$	1	✓	0	14	15	1.1	88		1 660		2 612 181
	2	✓				1.1					
	1	✓	11	42	46	16.5	80	57	1 520	854	2 398 239
	2	✓		92	102	63.3	61		1 125		1 765 110
	1	✓	14	118	130	153.7	71		1 342		2 123 382
	2	✓		234	262	317.4	57		949		1 468 171

Number of paths and feasible paths in the results obtained for *substring*

are shown in Table 6.3. The respective ratios of feasible paths over paths in the input LTS for $l = 30$ and 50 are approximately 0.072 and 0.01. We could not track the number of feasible paths for $l = 100$.

With *la* set to zero, both methods of abstraction produce the same LTS: the infeasibility of paths of the first and second group is discovered.

Both methods yield new LTS when *la* is set to 11 or above. This prevents a subsumption between the first two occurrences of 4, 4^0 and 4^1, that are separated by one iteration of the internal loop. Indeed, consider the sub-path $sp = 4 \cdot 5 \cdot 7 \cdot 8 \cdot 10 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 4 \cdot 8$ of length 11: it does one unsuccessful comparison in the internal loop before exiting it, then exits the external loop, enters it again and performs one successful comparison in the internal loop then exits it because $j = 1 = l_2$.

This sub-path is feasible from 4^0, since at this point, nothing is known about l_2: executing *sp* from 4^0 imposes l_2 to be 1, which is consistent with the end of *sp* (i.e. leaving the internal loop because $j = 1 = l_2$). However, if executed from 4^1, it imposes l_2 to be 2 because one iteration of the internal loop was needed to reach 4^1, which is in contradiction with the end of *sp*. This is the shortest sub-path that is feasible from 4^0 but infeasible from 4^1, which explains why the algorithm only produces new LTS when *la* reaches 11.

Since subsumption of 4^1 by 4^0 is prevented, the internal loop is unfolded again from there, which, combined with the fact that the same phenomenon occurs at other occurrences of 4, forces the algorithm to find different subsumptions than previously, yielding different and more accurate results.

Both abstraction methods rule out paths of the first and second groups. When removing constraints from path predicates, infeasibility of paths of the third group is discovered for $l_s = 0$. Updating stores performs best here: the property is discovered for $l_s = 0$, $l_s = 1$ and $l_s = 2$. In the latter case, the ratios of feasible paths over paths are 0.934 and 0.759 for $l = 30$ and $l = 50$.

Setting *la* to 14 allows to better over-approximate the set of feasible paths of *substring*: the first method of abstraction allows to detect paths of the third group for $l_s = 0$ and $l_s = 1$. The second method performs best: infeasibility of paths of the third group is discovered for $l_s = 0$, 1 and 2. Infeasibility of paths of the third group for $l_s = 3$ is almost discovered: the algorithm detects that at least three successful comparisons are necessary for $l_s = 3$, but fails to discover that a fourth successful comparison is needed. When removing constraints, the infeasibility of paths of the third group is discovered for $l_s = 0$ and 1. Paths of the first and second group are systematically detected infeasible.

The value of *la* might seem rather high compared to the value of l when the latter is set to 30 (*la* approaching the half of l). We recall that, unlike *la*, l is not a parameter of our approach but only the (maximal) length of paths we want to count to compare the LTS produced by our approach. Each $\mathcal{S}'$

described in the different tables of this chapter is built only once: one can then count or draw paths in each $\mathcal{S}'$ as many times as needed, for values of l as large as one wants.

The *substring* example shows that our approach can be used to over-approximate sets of feasible paths even in cases where those sets are not regular languages: in the best case, described by the last line of Table 6.3, every infeasible path of length at most 30 has been eliminated, and the ratio of feasible paths over paths for $l = 50$ is now approximately 0.9 (but we were not able to track it for $l = 100$).

6.1.4 Bubble Sort

We now present some results obtained for the *bubble sort* program. Its pseudocode and LTS are given in Figure 6.7 and 6.8, respectively. Similarly to the example of *merging sort*, instructions that use or set the values of the input array a have no influence on the feasibility of paths: we use a slightly simplified version of the LTS for our analysis.

```
Function bubble(int[] a, int i, int l)
1   let swap = 1;
2   while swap ≠ 0 do
3       swap ← 0;
4       let i = 1;
5       while i < l do
6           if a[i − 1] > a[i] then
7               let tmp = a[i − 1];
8               a[i − 1] ← a[i];
9               a[i] ← tmp;
10              swap ← 1;
11          i ← i + 1;
```

Figure 6.7: The bubble sort program.

The LTS of Figure 6.8 presents three groups of infeasible paths:

- paths that reach the end of the program while *swap* equals 1,
- paths that enter the external loop while *swap* equals 0,
- paths that do not go through the same number of iterations of the internal loop at each iteration of the external one.

Because of paths of the third group, the set of feasible paths of the *bubble sort* LTS is not a regular language. Number of paths and feasible paths for *bubble sort* are given in Table 6.4.

When la is set to zero, both methods of abstraction return the same LTS: only the path that never enter the outer loop is detected infeasible. Both methods produce new LTS when la is set to 2. Once again, updating

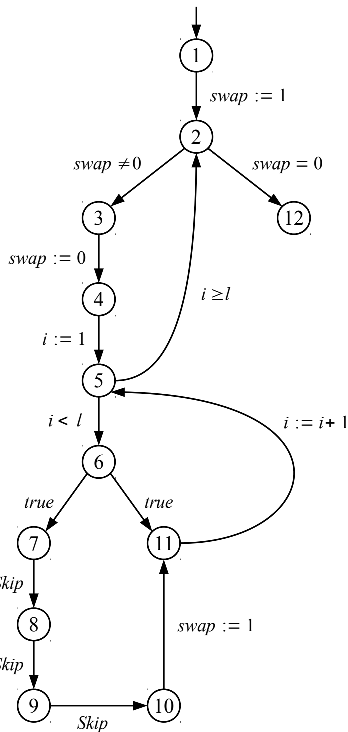


Figure 6.8: The bubble sort LTS.

stores yield the best results here: paths of the first and second groups are eliminated, while only those of the first group are detected with constraints removal. This is due to the fact that, at some point during the analysis, an abstraction of the configuration of an occurrence of vertex 2 requires to remove a constraint relating *swap* to its value 0.

Results obtained by removing constraints can be improved by setting *la* to 8: paths of the first group are still removed, but this time the algorithm also rules out some paths of the second group as well as those of the third group that perform exactly one iteration of the internal loop.

Updating stores yield the same results for $la = 3$, but a chain of refinements occurs when *la* is set to 4, which prevents to build a complete red part. This is due to some abstractions that are performed at occurrences of vertex 4 that require to forget the concrete value of i at those point, which causes paths that do not perform the same number of iterations of the internal loop at each passage through the external one to be considered feasible. This phenomenon is repeated infinitely, which forces to either bound the length of paths or to disable refinements. The sixth line of Table 6.4 describes the result obtained with refinements allowed and *mrl* set to 30: the red part is not complete and, once again, the accuracy of $\mathcal{S}'$ greatly

Table 6.4: Paths (P) and feasible paths (FP) in *bubble sort*.

	a	r	la	$\|L\|$	$\|\Delta\|$	t	$l = 30$		$l = 50$		$l = 100$
							P	FP	P	FP	P
$\mathcal{S}$				12	14		494		76 625		2.4×10^{10}
$\mathcal{S}'$	1	✓	0	16	18	2.7	493		76 624		2.4×10^{10}
	2	✓				3.7					
	1	✓	2	40	48	7.2	337		52 171		1.7×10^{10}
	2	✓		22	25	6.7	60	13	3 530	82	1×10^{8}
	1	✓	8	73	84	48	121		9 569		6×10^{8}
	2	✓	4	180	193/13	116.6	26		3 316		9.2×10^{8}
				35	39	7	60		3 530		1×10^{8}
			8	76	85	23.9	60		3 408		8.8×10^{7}

decreases with l.

The fifth line shows the LTS obtained with the previous parameters but with refinements disabled: once again, this comes at the price of introducing looser abstractions, which yield a LTS that is less precise than in the previous case for $l = 30$, but a lot more as l grows.

The last line describes the LTS obtained with the same parameters than at the fifth line, but with la set to 8: although refinements are disabled, this is the most precise result with a complete red part that we obtain for *bubble sort*.

As said previously, the set of feasible paths of *bubble sort* is not a regular language, as is the case with *substring*. However, the algorithm performs best on the latter than on the former. We believe this is due to the fact that the two loops of *substring* are governed by two distinct indexes, while those of *bubble* are only governed by i, which induces a chain of refinements when la is set to 4 or above. We think that this chain of refinements could be avoided by using more accurate abstraction methods, that would learn from safeguard conditions, for example. Nonetheless, *bubble sort* clearly reveals some limitations of our approach in its current state.

6.1.5 Bounded Loops

Our approach was designed specifically to handle unbounded loops, but it is legitimate to wonder how it behaves in presence of bounded loops. In this section, we consider three variations of a simple homemade example, *bloop*, whose pseudocode and LTS are given in Figures 6.9 and 6.10. This program takes four integers, *start*, n, x and y, as inputs and performs n iterations of its loop, during which two actions can be done depending on the fact that x equals y or not. Although the loop of this program in itself is not bounded, we analyze it with three different preconditions for the value of n that simulate a bound: $n \leq 2$, $n = 2$ and $n \geq 2$. Instead of using a constant

for the initial value of the loop index, we use a parameter *start*.

```
Function bloop(int start, int n, int x, int y)
1   let i = start;
2   while i < start + n do
3       if x = y then
4           skip;
        else
5           skip;
6       i ← i + 1;
```

Figure 6.9: The *bloop* program.

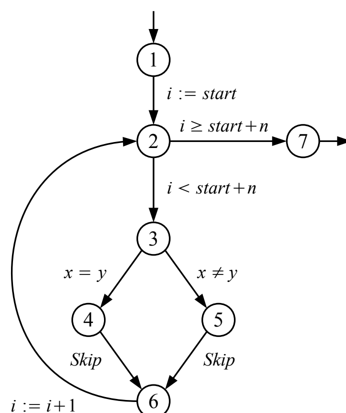


Figure 6.10: The *bloop* LTS.

The conditional block at line 3 was added to introduce a number of infeasible paths in *bloop* and see how the algorithm reacts: the truth value of $x = y$ does not change during the execution of *bloop*, and the algorithm should detect that, after the first iteration of the loop, only one path can be taken in each subsequent iteration [2]. Let k be a number of iteration of the loop of *bloop*: complete paths of *bloop* are all of length $4k + 2$. The LTS of Figure 6.10 contains exactly 2^k paths of length $4k + 2$, but only 2 of them are feasible. The following results show that our algorithm both detect subsumptions that are consistent with the three preconditions and is able to rule out infeasible paths due to the internal condition.

If *pre*, the precondition parameter of our algorithm, is set to $n \leq 2$, then *bloop* can only terminate before entering its loop, or after doing no more than one or two iterations. In this case, if refinements are allowed but the look-ahead depth set to 0, both methods of abstractions yield the red part depicted in Figure 6.11.

[2] This example is inspired from an industrial case study reported in [27]

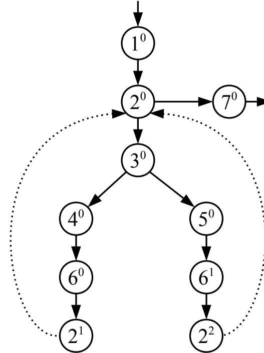


Figure 6.11: A complete red part of the *bloop* example with $n \leq 2$ for $la = 0$.

Indeed, the shortest sub-path *sp* starting at the first occurrence of 2 and leading to an occurrence of 7 is already feasible if $n \leq 2$: the two abstractions that are performed from this point do not trigger the counterexample guided refinement mechanism and are never refined. This also causes the infeasibility of paths due to the truth value of $x = y$ to be undetected. Since refinements cannot help, the only choice is to set *la* to a non-zero value. Both sub-paths in the body of the loop being made of four transitions, a look-ahead depth of 5 (4 for the loop body, plus 1 to discover the infeasible guard) would allow to discover that doing a third iteration of the loop is not possible here, and would prevent the two subsumptions. However, setting *la* to 2 is enough: this allows to detect that, after the first iteration, only one sub-path of the loop body is still feasible, i.e. that the truth value of $x = y$ has been set once and for all. This also prevents the two subsumptions: both abstraction methods yield the red part of Figure 6.12, with *la* set to 2 and refinements disabled.

If we consider the precondition $n = 2$, then *bloop* must terminate after exactly two iterations of its loop. When *la* is set to 0, the analysis starts as in the previous case: the first two possible subsumptions of Figure 6.11 are established, but then are both refined, since this time they actually turned *sp* into a feasible path. This causes the loop to be unfolded once more along the two paths. The same phenomenon occurs along both: the algorithm subsumes the third occurrence of 2 by the second, which triggers a refine-and-restart phase, causing the loop to be unfolded a third time along each path. From there, the algorithm discovers that the loop cannot be entered again and that *bloop* must terminate. Again, both methods of abstraction yield the same result. Disabling refinements and setting *la* to 2 yields the same result but it is obtained much faster: it takes 4.5 seconds to build the red part depicted in Figure 6.13 with refinements enabled and *la* set to 0, but only 1.1 seconds without refinements but *la* set to 2. This is due to the fact that refinements require rebuilding large parts of the red part.

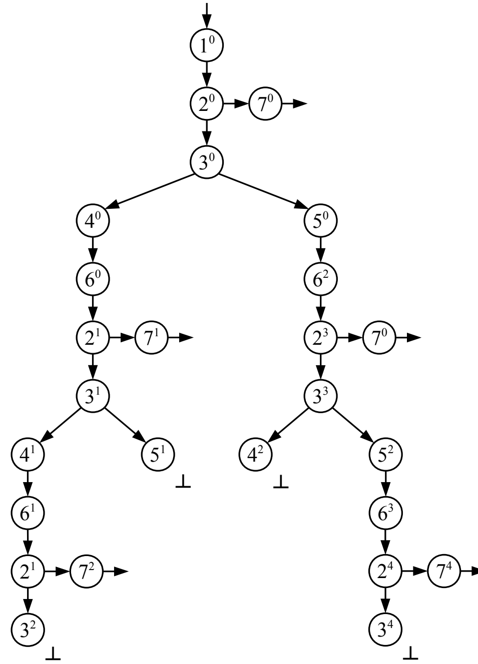


Figure 6.12: A complete red part of the *bloop* example with $n \leq 2$ for $la = 2$.

Finally, if *pre* is $n \geq 2$, *bloop* can only terminate after doing at least 2 iterations of its loop. With both methods of abstraction, and with refinements allowed or *la* set to 2, the algorithm yields the red part of Figure 6.14 (modulo a renaming of vertices). The analysis goes on exactly as in the previous case, the only difference being that, along each path, the loop can be entered more than twice. This causes the algorithm to detect the two subsumptions between the fourth and third occurrences of 2 along each path and end the analysis. Once again, the analysis is much faster when refinements are disabled than enabled (2.1 seconds vs. 6).

As illustrated by this example, our algorithm is also efficient in presence of bounded loops: in each previous case, our approach detected the infeasibility of paths related to the bound of the loop as well as to the truth value of the condition of the internal block. This example also illustrates the facts that the look-ahead mechanism usually yield results faster and might compensate for the absence of refinements, or even perform better on some examples.

6.1.6 Modulo Example

We still consider the *loop* program, but now analyze it with the precondition that $start = 0$ and n is even, denoted $n = 0[2]$. As a result, feasible paths

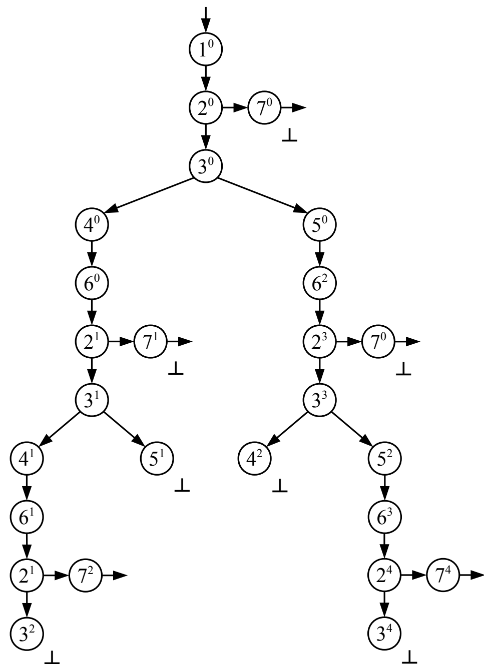


Figure 6.13: A complete red part of the *bloop* example with $n = 2$.

of the LTS of Figure 6.10 can only exit the loop after having performed an even number of iterations of the loop. Both methods of abstractions yield the same results in all cases described below.

Once again, allowing refinements but setting the look-ahead depth to 0 does not allow to detect any infeasible paths and yields the LTS of Figure 6.10.

Contrary to the other preconditions of *bloop*, allowing refinements and setting *la* to 2 causes an infinite chain of refinements. The reason for this behaviour can be easily understood when considering the red part depicted in Figure 6.14 obtained when refinements are not allowed.

As can be seen, the look-ahead mechanism drives the algorithm to only establish subsumption links whose ends are separated by two iterations of the loop; along each path, the subsumption between the first and third occurrences of 2 are prevented because the truth value of $x = y$ is still unknown during the first iteration of the loop. However, none of the occurrences of the final location 7 are marked: the abstractions performed at 2^1 and 2^4 both require to forget the value of i and the fact that it is odd, i.e. that exactly one iteration of the loop was performed to reach those two vertices. As a result, the algorithm is not able to detect that the execution cannot leave the loop from there. Allowing refinements in this case would simply postpone those abstractions along each path, which would then be refined

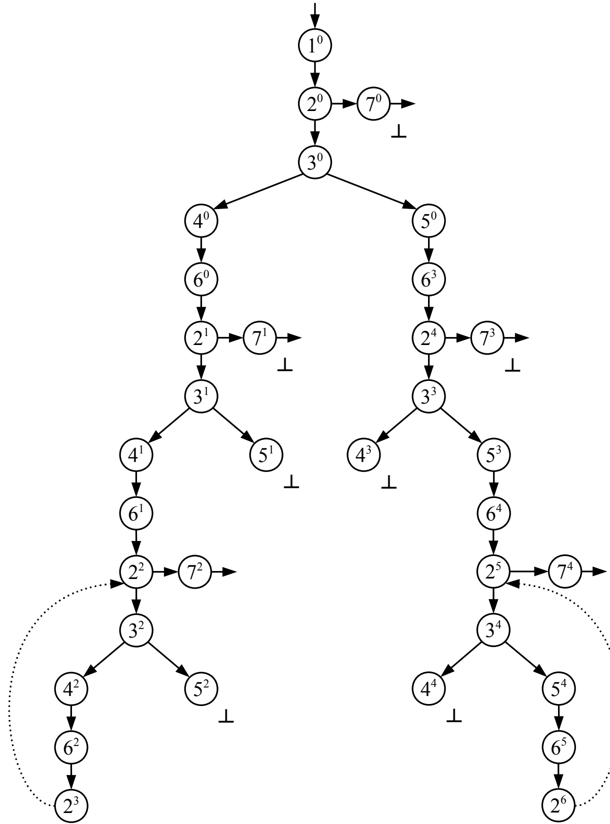


Figure 6.14: A complete red part of the *bloop* example with $n \geq 2$.

and so on, causing the infinite chain mentioned earlier.

The only possible way to discover the infeasibility of paths leaving the loop after an odd number of iterations of the loop would be to somehow infer the fact that i is odd at 2^1 and 2^4. Since our methods of abstraction only remove constraints or disconnect program variables from path predicates, this kind of guessing is out of the reach of our approach in its current state. Here, our algorithm only discovers that the result of $x = y$ remains constant and only discards infeasible paths related to this condition.

6.2 Discussions and Possible Improvements

6.2.1 Extending Refinements

A first possible way we see to augment the infeasible path detection power of our approach could come from observing the interaction between the propagation of abstractions and the counterexample guided refinement mechanisms. The latter is an *a posteriori* method for controlling abstractions

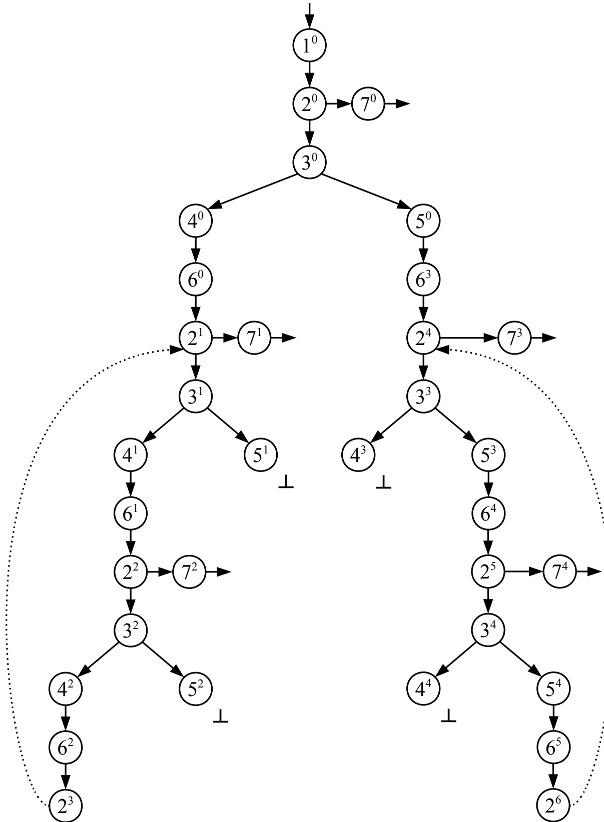


Figure 6.15: A complete red part of the *bloop* example with $start = 0$ and $n = 0[2]$.

that consists in refuting them if they introduce some infeasible paths in the red-black graph being built. These checks are based on individual paths, since we do not consider subsumption links when searching for faulty abstractions. As an effect, the infeasible path elimination power of this control mechanism is somewhat limited: abstractions that only introduce infeasible paths going through subsumption links cannot currently be detected by this approach. The scope of this mechanism could be extended by not only considering red paths that do not go through subsumption links, but also those going exactly once through such links. It would not be necessary to consider more passages through subsumption links: the goal is to find a configuration that turned an infeasible path into a feasible one, but the same stacks of configurations would be checked multiple times along cycles. However, we believe that this might:

- at best have a major (negative) impact on performances since the number of paths to consider might be much larger, and since each

abstraction that is refined causes the analysis to restart. Moreover, since faulty abstractions might be detected long after having been introduced, refine-and-restart phases usually requires re-building large parts of the red-black graph,

- at worst considerably increase the chances of infinite chains of refine-and-restart occurring if we do not have at the same time a way to compute more precise abstractions by learning from safeguard conditions.

As we mentioned earlier, being able to compute accurate abstractions considering safeguard conditions is a major improvement that might allow to avoid some chains of refinements, which is, in our opinion, the main limitation of our approach at the moment.

Note that refinements are always triggered after the propagation of some abstraction caused a red vertex to be unmarked or not to be marked when it should have been: one might think that a possible solution would be to simply not allow propagating such abstractions (i.e. procedure `propagable` would return *false* when reaching a marked vertex). Since no refinements would ever be needed, this would actually make the counterexample guided refinement mechanism pointless. But this is forgetting that this mechanism also yields safeguard conditions from which one could take advantage.

Since we did not implement at the moment any solution to compute abstractions by learning from such conditions and for the reasons exposed above, we also did not implement this possible extension of the counterexample guided refinement mechanism.

6.2.2 Look-Ahead Mechanism

Interpretation of the Look-Ahead Depth

A simple improvement one might add to the look-ahead mechanism lies in the interpretation of the *la* global parameter. At the moment, we interpret it as the length of feasible sub-paths starting at potential subsumees and subsumers, but this is an arbitrary decision and one might interpret it differently. For example, *la* might be interpreted as the k factor of the so-called k*-path* coverage criterion used in white-box path-based testing (see [52] for example). In this case, we would compare sets of feasible sub-paths that basically go at most *la* times through each reachable loop of the program, rather than sets of feasible sub-paths of length at most *la*.

Note that this would not change drastically the behavior of the overall approach, but only potentially drive the algorithm towards different subsumptions than in its actual version. This feature was implemented in preliminary versions of our implementation, but those versions were not based on the formalization presented in Chapter 4. Although we did not experiment this approach with our current implementation and plan to do so in

the near future, we believe that doing so would not yield very different results than those presented in this chapter since this would only push our current approach a little further but still relying on the same basis.

Look-Ahead and Abstraction

As said in 5.3.5, the look-ahead mechanism only provides hints about potentially more accurate subsumptions to the algorithm, but is not involved in computing abstractions. Since both current abstraction methods proceed in a given order without taking advantage of safeguard conditions, the look-ahead mechanism, if enabled, does not guarantee that the resulting LTS will be more accurate, but only drives the search for subsumptions in potentially more rewarding regions of the set of paths of the input LTS.

Since look-ahead checks are done before performing abstraction and since the look-ahead mechanism does not help computing better abstractions, it might not prevent introducing some infeasible paths in the red part once a subsumption has been established, if some abstraction was required. Once again, this is a direct consequence of the propagation of abstractions causing red vertices to be unmarked. One might think that a possible naive and expensive solution would be to check, after an abstraction has been propagated, that sets of feasible sub-paths starting at both ends of the new subsumption link have not been modified, but this would simply be equivalent to not allow `propagate` to unmark red vertices and we saw earlier that this might totally prevent the algorithm to establish any subsumption and build complete red parts in most cases.

One possible way of taking advantage of the information gathered while doing look-ahead checks might be to compute safeguard conditions from the infeasible sub-paths that might be met during the check, but this obviously assumes methods of abstraction that would provide more accurate results considering such conditions. This would not make the counterexample guided refinement useless since the value of *la* might not allow to detect that an abstraction turned an infeasible path into a feasible one: this might only happen after *la* steps, and some faulty abstractions might still be refined a posteriori.

6.2.3 Abstraction methods

We said in Chapter 3 that the way abstractions are performed is a crucial point in our approach. We also mentioned several times that a possible way to improve the infeasible path detection power of our approach might consist in using abstraction methods that yield more and more accurate abstractions as red vertices are labeled with safeguard conditions. Besides directly leading to better over-approximations of sets of program states at red vertices and thus feasible paths, this might also indirectly improve the

results of our approach by preventing some infinite chains of refinements and help produce complete red parts in some cases. We do not propose any solution for this at the moment. A simple approach might be to add some backtracking in our abstraction methods. For example, as soon as entailment is lost because a constraint c is removed from the path predicate p of the configuration to abstract, the abstraction procedure might go one step back and restore c, before considering other elements of p as targets for removal. The same idea can be applied to program variables when updating stores, but this might be less efficient than in the previous case, since abstracting a program variable usually disconnects it from more than one constraint of p.

Although adapting our two current methods of abstraction to learning from safeguard conditions is definitely worth investigating, we remark that those two methods also have inherent limitations, as illustrated by our last example presented in 6.1.6. In that case, the fact that the crucial information regarding the feasibility of paths that exit the loop or not could not be retrieved by our current methods is not related to them not taking advantage of safeguard conditions, but to their very nature.

6.2.4 Subsumptions Between Different Paths

In previous examples, we only considered subsumption links whose ends occur along the same symbolic paths, but our algorithm was designed to handle both subsumptions between red vertices that occur along a same or different symbolic paths. Indeed, our definition of subsumption and our methods for computing and propagating abstractions applies in both cases. However, we observe that simply applying our algorithm as introduced in Chapter 5 when considering subsumptions between different paths usually yields significantly less accurate LTS than those obtained when considering only subsumption along the same symbolic paths. Since our goal is to detect as many infeasible paths as possible, we chose to first introduce results obtained in the most favorable cases — i.e. considering subsumption along unique paths — then discuss what happens when considering subsumptions between different paths, and how to handle those subsumptions in order to maximize the infeasible path detection power of our approach. We see three possible ways to handle these subsumptions.

- The first one is simply to apply our algorithm as described in Chapter 5, i.e. given a red vertex rv to subsume and rv' a potential subsumer for rv, the algorithm will first compare sets of feasible sub-paths starting at rv and rv' (if la is not zero), check for a natural subsumption between the two, and finally search for a suitable abstraction of rv', and so on. The accuracy of resulting LTS usually drops in most cases compared to LTS obtained when only considering subsumption along the unique symbolic paths, since more candidates are considered and

since each of them might be abstracted to force a subsumption.

- The second possibility consists in not allowing potential subsumers occurring on a different path than the potential subsumees to be abstracted. The process of detecting a subsumption would be exactly the same than previously, except for the fact that, if the potential subsumee and subsumer lie on different symbolic paths, then the algorithm only checks for natural subsumption and never tries to force the subsumption through abstraction. Without surprise, LTS obtained in this manner are usually more accurate than in the previous case, since a number of abstractions will be immediately discarded. However, we notice than those LTS are still less accurate than those obtained when only considering subsumption along unique symbolic paths. This is naturally due to subsumption being defined as an inclusion and not an equality: as said previously in this document, each subsumption possibly lessens the accuracy of the resulting LTS.

- The third way to handle subsumptions between different paths we see consists in only searching for subsumption along unique symbolic paths, as we did in the previous examples, and search for subsumption links between different paths only after the red part has been built. In this case, a subsumption link would be established between two different occurrences rv and rv' of a given black vertex if and only if the red sub-graphs rooted at rv and rv' are isomorphic. Since this part of the analysis would only take place after the actual detection of infeasible paths, the LTS obtained in this case would be as precise as those presented in the previous section but more compact.

As said previously, both first and second methods for handling subsumptions between different paths have the inconvenient that they yield less accurate LTS than those obtained when only considering subsumptions along unique paths. They however share one advantage over the third: they yield results faster than in the case of subsumptions along unique paths. This is due to the fact that paths might be merged, mitigating the explosion of the number of paths that usually occurs during symbolic execution and also because computing graphs isomorphisms is naturally a costly operation (although in our case, not all possible pairs of vertices would have to be considered, but only those corresponding to the same black vertices). In terms of infeasible path detection power, the third approach is obviously the better, however, since the Rukia Library can efficiently draw paths in graphs with billions of vertices, one could argue that searching for isomorphic sub-graphs in red parts once they have been built might not be of great interest in most cases. As a result, we do not give a definitive answer as how to handle this particular type of subsumptions: we think that, as with

the other heuristics that were presented earlier in this document, the best choice might simply depend on the program under analysis.

6.3 Summary

In this chapter, we have described a number of experiments and their results in order to assess the infeasible path detection power of our approach. Although the heuristics we currently use to guide our algorithm are still simple, those experiments have, most of the time, led to promising results. In a number of cases, our algorithm was able to detect all infeasible paths contained in the input LTS. Even in the least favorable cases, i.e. examples whose sets of feasible paths are not regular languages, our approach yields LTS that present far better ratios of feasible paths over paths, despite the fact that the set of feasible paths can only be over-approximated in such cases.

These experiments highlight some limitations of our approach in its current state and, consequently, give some indications of the possible ways this approach might be improved. As mentioned several times in this document and as illustrated by the previous examples, the way abstractions are performed seems to be the crucial point to investigate in order to improve the infeasible path detection capability of the overall approach. We imagine two orthogonal ways to improve the accuracy of the abstraction process: *(i)* use abstraction methods that will take advantage of safeguard conditions to produce finer results and *(ii)* using abstraction methods that not only remove information from path predicates but also infer new facts. As said previously, we do not yet propose solutions for these two problems, but we plan to investigate in these directions.

Chapter 7

Conclusion

In this thesis, we address the problem of graph transformations that discard infeasible paths, but still preserve the behavior of the program. Our motivation comes from random structural biased testing. Our approach produces good over-approximations of the set of feasible paths of the program under analysis: this results in new graphical representations of programs that are more detailed than the original control flow graphs. Our objective is to facilitate drawing feasible paths and thus to produce test cases in the context of random structural biased testing. In Chapter 2, we introduced the notion of random walks, which allows to explore graphs of programs and to draw some of their paths, and illustrated why and how testing approaches relying on such random methods were limited by the existence of infeasible paths. We also presented a number of recent works addressing the problems of infeasible path detection and symbolic execution in presence of unbounded loops.

In Chapter 3, we presented the theoretical notions our approach relies on, and introduce the concept of red-black graph — the data structure that is at the center of our approach — as well as their transformations that express the basic operations performed by our algorithm.

Our algorithm is based on classical symbolic execution, detection of subsumption, abstraction, counterexample guided refinements, and is driven by a number of heuristics in order to improve its infeasible path detection power. These features interact in a rather intricate manner, which makes our approach a natural candidate for machine aided verification. We presented in Chapter 4 the formal description of the graph transformations, built using the Isabelle/HOL interactive theorem prover, in order to prove the key properties of our approach, namely that it is correct and that it preserves the set of feasible paths of the original control flow graph.

We described our algorithm in full details in Chapter 5, and illustrated, step by step, how it behaves and how its main features and heuristics interact on a typical example.

Finally, in order to assess the infeasible path detection power of approach, we reported in Chapter 6 various experiments, and commented and interpreted the results obtained. These results are promising: in the most favorable cases, our algorithm allowed to discover all the infeasible paths of examples with multiple loops, and to greatly improve the ratios of feasible paths over paths even in the least favorable ones. These experiments also revealed some limitations of the approach in its current state, but this also points us to some indications on how our algorithm could be improved.

The works reported in this thesis could not only help in the context of random structural biased testing, but could also help to improve the accuracy of any other white-box testing approach or software analysis technique based on control flow graphs or equivalent representations of programs, such as static analysis or worst-case execution time analysis, for example.

We now evoke various perspectives for our works.

Learning from Safeguard Conditions

Throughout this document, we mentioned several times that a crucial point in our approach lies in the way abstractions are performed. Obviously, more accurate abstractions will lead to a better infeasible path detection power and investigating in this direction is definitely worth it.

Our approach is currently limited by the fact that our methods of abstraction do not learn from safeguard conditions: such conditions allow to prevent too crude abstractions, but does not actively participate in finding more accurate abstractions. As illustrated by the examples given in this thesis, our counterexample guided refinement mechanism can help detect more infeasible paths while yielding complete red parts, but it can also cause some infinite chains of refinements to occur, preventing to build complete red parts. We conjecture that methods of abstraction that would take advantage of safeguard conditions in order to provide more accurate abstractions might prevent some of these infinite chains, as well as increase the overall infeasible path detection power of our approach.

We also observe that, if we were to use such abstraction methods, then we might possibly take advantage of some user-given or automatically inferred invariants that we would inject as safeguard conditions in our analysis. Nothing prevents us from doing so at the moment, but we would be in the same situation than previously: our abstraction methods are not able to take advantage from such invariants.

We definitely plan to investigate in this direction in the future.

Extending the Input Language

One of the limitations of our approach in its current state is that its scope is limited to programs containing simple statements. For example, we do

not support arrays nor pointer expressions. Obviously, extending the input language of our approach to such constructs would allow us to assess the infeasible path detection power of our approach on more practical and concrete examples.

This would require using a far more advanced type of memory model — such as those presented in [10] or [49] — than we currently do: at the moment, our memory model is a configuration, i.e. a couple made of a mapping from program to symbolic variables and a conjunction of constraints over those. Trìk and Strejcek presented in [49] a segment-offset-plane memory model that allows to handle allocation, read, write, deallocation and test for memory initialization in the context of symbolic execution. The authors also provide some directions for handling manipulation of composed objects or unions.

We think that, although not an easy task, our approach could be adapted to such a memory model without changing the basic ideas it relies on. Handling these new language constructs will require new forms of labels. We conjecture that the part of our approach that will need the most re-thinking is the way abstraction are performed.

As another example, our approach is only intraprocedural at the moment. We think that it could be extended to the interprocedural case by computing procedure summaries, as is done in [40] using lazy annotation. We recall that our approach is very close by nature to the one presented in that paper and to lazy annotation, and that extending our approach to the interprocedural case might be done relatively easily.

Integration to Static Analysis Tools

One of our objectives for the future is to integrate our approach to an existing static analysis tool. The interest might be twofold: first, the results provided by our approach could improve the performance or accuracy of the techniques already implemented in the tool, but, on the other hand, this might be an opportunity for us to imagine new heuristics for our approach that take advantage of said existing techniques. Our approach was implemented in a prototype, coded in Ocaml, that we plan to integrate to the Frama-C platform in the near future.[1]

[1] The documentation and a number of publications about the Frama-C platform can be found at `https://frama-c.com/`.

Bibliography

[1] Romain Aïssat, Marie-Claude Gaudel, Frédéric Voisin, and Burkhart Wolff. Pruning infeasible paths via graph transformations and symbolic execution: a method and a tool. Technical Report 1588, L.R.I., Univ. Paris-Sud, 2016.

[2] Romain Aïssat, Marie-Claude Gaudel, Frédéric Voisin, and Burkhart Wolff. Pruning infeasible paths via graph transformations and symbolic execution: a method and a tool. In *Software Quality, Reliability and Security - 2nd International Conference, QRS 2016, Vienna, Austria, August 1-3, 2016, Proceedings*, 2016.

[3] Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible paths elimination by symbolic execution techniques - proof of correctness and preservation of paths. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2016.

[4] Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible paths elimination by symbolic execution techniques: Proof of correctness and preservation of paths. *Archive of Formal Proofs*, August 2016. `http://isa-afp.org/entries/InfPathElimination.shtml`, Formal proof development.

[5] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, 1996*, pages 102–107, Jun 1996.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[7] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivani, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *In SAS, volume 5079 of LNCS*, pages 238–254, 2008.

[8] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 97–103, New York, NY, USA, 2001. ACM.

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[10] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A precise and abstract memory model for C using symbolic values. In Jacques Garrigue, editor, *Programming Languages and Systems: 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 449–468. Springer International Publishing, 2014.

[11] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[12] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 307–321, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[13] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, November 1997.

[14] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[15] P. Bourque and eds. R.E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society, 2014.

[16] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE'2008*, pages 443–446. IEEE, 2008.

[17] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[18] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[19] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 215–224. IEEE, 2010.

[20] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased random exploration of large models and application to testing. *Int. Journal on Software Tools for Technology Transfer*, 14(1):73–93, 2011.

[21] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 12–22, New York, NY, USA, 2004. ACM.

[22] Lydie du Bousquet. A new approach for software testability. In Leonardo Bottaci and Gordon Fraser, editors, *Testing - Practice and Research Techniques, 5th International Academic and Industrial Conference, TAIC PART 2010, Windsor, UK, September 3-5, 2010. Proceedings*, volume 6303 of *Lecture Notes in Computer Science*, pages 207–210. Springer, 2010.

[23] Lydie du Bousquet, Farid Ouabdesselam, and Jean-Luc Richier. Expressing and implementing operational profiles for reactive software validation. In *Ninth International Symposium on Software Reliability Engineering, ISSRE 1998, Paderborn, Germany, November 4-7, 1998*, pages 222–230. IEEE Computer Society, 1998.

[24] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 227–236, New York, NY, USA, 2005. ACM.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[26] Allen Goldberg, T. C. Wang, and David Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 80–94, New York, NY, USA, 1994. ACM.

[27] Sandrine-Dominique Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique. (Using Combinatorial Structures for Statistical Testing).* PhD thesis, University of Paris-Sud, Orsay, France, 2004.

[28] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[29] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. of the 31st ACM Symp. on Principles of Programming Languages*, POPL '04, pages 232–244. ACM, 2004.

[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[32] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.

[33] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2005.

[34] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 758–766. Springer, 2012.

[35] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded symbolic execution for program verification. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 396–411, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[37] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[38] N. Malevris, D. F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *J. Electron. Mater.*, 19(6):115–118, June 1990.

[39] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[40] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Int. Conf. on Computer Aided Verification, CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[41] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[42] John Musa, Gene Fuoco, Nancy Irving, Diane Kropfl, and Bruce Juhlin. Handbook of software reliability engineering. chapter The Operational Profile, pages 167–216. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[43] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 215–224, New York, NY, USA, 2007. ACM.

[44] Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015.

[45] Johan Oudinet. *Approches combinatoires pour le test statistique à grande échelle*. PhD thesis, Université Paris-Sud XI, Ph. D. thesis, 2010.

[46] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[47] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[48] Pascale Thévenod-Fosse and Hélène Waeselynck. An investigation of statistical software testing. *Softw. Test., Verif. Reliab.*, 1(2):5–25, 1991.

[49] Marek Trtík and Jan Strejcek. Symbolic memory with pointers. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2014.

[50] Silvia Regina Vergilio, José Carlos Maldonado, and Mario Jino. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. *Journal of the Brazilian Computer Society*, 12(1):73–88, 2006.

[51] Daniel Wasserrab. Information flow noninterference via slicing. *Archive of Formal Proofs*, March 2010. Formal proof development.

[52] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Dependable computing - edcc 5: 5th european dependable computing conference, budapest, hungary, april 20-22, 2005. proceedings. pages 281–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[53] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

Appendix A

Isabelle/HOL Formalization

A.1 Introduction

This appendix is excerpted from [4], the proof script registered in the "Archive of Formal Proofs" server.

We proceed as follows (see Figure A.1 for the detailed hierarchy). First, we formalize all the aspects related to symbolic execution, subsumption and abstraction (`Aexp.thy`, `Bexp.thy`, `Store.thy`, `Conf.thy`, `Labels.thy`, `SymExec.thy`). Then, we formalize graphs and their paths (`Graph.thy`). Using extensible records allows us to model Labeled Transition Systems from graphs (`Lts.thy`). Since we are interested in paths going through subsumption links, we also define these notions for graphs equipped with subsumption relations (`SubRel.thy`) and prove a number of theorems describing how the set of paths of such graphs evolve when an arc (`ArcExt.thy`) or a subsumption link (`SubExt.thy`) is added. Finally, we formalize the notion of red-black graphs and prove the two properties we are mainly interested in (`RB.thy`).

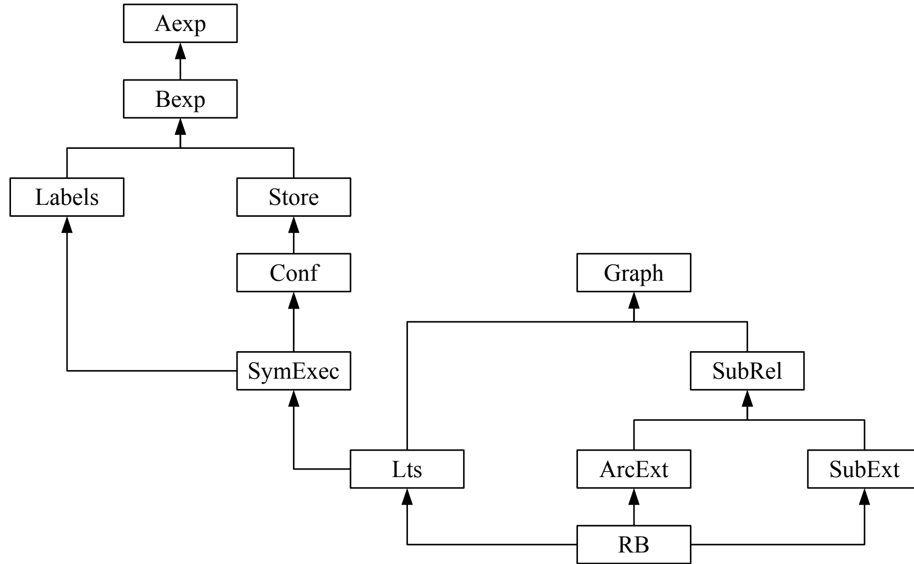


Figure A.1: The hierarchy of theories.

theory *Aexp*
imports *Main*
begin

A.2 Arithmetic Expressions

In this section, we model arithmetic expressions as total functions from valuations of program variables to values. This modeling does not take into consideration the syntactic aspects of arithmetic expressions. Thus, our modeling holds for any operator. However, some classical notions, like the set of variables occurring in a given expression for example, must be rethought and defined accordingly.

A.2.1 Variables and their domain

Note: in the following theories, we distinguish the set of *program variables* and the set of *symbolic variables*. A number of types we define are parameterized by a type of variables. For example, we make a distinction between expressions (arithmetic or boolean) over program variables and expressions over symbolic variables. This distinction eases some parts of the following formalization.

Symbolic variables. A *symbolic variable* is an indexed version of a program variable. In the following type-synonym, we consider that the abstract type $'v$ represent the set of program variables. The set of program variables can be interpreted as the set of variables of a given program, excluding all variables of other programs, or as the set of all variables of any program. Within Isabelle/HOL, nothing is known about this set. It might be infinite or not: it is never needed to assume the former or the latter. On the other hand, the set of symbolic variables is infinite, independently of the fact that the set of program variables is finite or not.

type-synonym $'v$ *symvar* $=$ $'v \times$ *nat*

lemma
$\neg$ *finite* (*UNIV*::$'v$ *symvar set*)
by (*simp add* : *finite-prod*)

The previous lemma has no name and thus cannot be referenced in the following. Indeed, it is of no use for proving the properties we are interested in. In the following, we will give other unnamed lemmas when we think they might help the reader to understand the ideas behind our modeling choices.

Domain of variables. We call D the domain of program and symbolic variables. In the following, we suppose that D is the set of integers.

A.2.2 Program and symbolic states

A state is a total function giving values in D to variables. The latter are represented by elements of type $'v$. Unlike in the $'v$ *symvar* type-synonym, the type $'v$ stands here for program variables as well as symbolic variables. States over program variables are called *program states*, and states over symbolic variables are called *symbolic states*.

type-synonym $('v,'d)$ *state* $=$ $'v \Rightarrow\ 'd$

A.2.3 The *aexp* type-synonym

Arithmetic (and boolean, see `Bexp.thy`) expressions are represented by their semantics, i.e. total functions giving values in D to states. This way of representing expressions has the benefit that it is not necessary to define the syntax of terms (and formulae) appearing in program statements and path predicates.

type-synonym $('v,'d)$ *aexp* $=$ $('v,'d)$ *state* $\Rightarrow\ 'd$

In order to represent expressions over program variables as well as symbolic variables, the type synonym *aexp* is parameterized by the type of variables. Arithmetic and boolean expressions over symbolic variables are used

to represent the constraints occurring in path predicates during symbolic execution.

A.2.4 Variables of an arithmetic expression

Expressions being represented by total functions, it makes no sense to say that a given variable is occurring in a given expression. We define the set of variables of an expression as the set of variables that can actually have an influence on the value associated by an expression to a state. For example, the set of variables of the expression $\lambda\sigma.\ \sigma\ x\ -\ \sigma\ y$ is $\{x,\ y\}$, provided that x and y are distinct variables, and the empty set otherwise. In the second case, this expression would evaluate to 0 for any state σ. Similarly, an expression like $\lambda\sigma.\ \sigma\ x\ *\ 0$ is considered as having no variable as if a static evaluation of the multiplication had occurred.

```
definition vars ::
  ('v,'d) aexp ⇒ 'v set
where
  vars e = {v. ∃ σ val. e (σ(v := val)) ≠ e σ}


lemma vars-example-1 :
  fixes e::('v,integer) aexp
  assumes e = (λ σ. σ x − σ y)
  assumes x ≠ y
  shows   vars e = {x,y}
unfolding set-eq-iff
proof (intro allI iffI)
  fix v assume v ∈ vars e

  then obtain σ val where e (σ(v := val)) ≠ e σ unfolding vars-def by blast

  thus v ∈ {x, y} using assms by (case-tac v = x, simp, (case-tac v = y, simp+))
next
  fix v assume v ∈ {x,y}

  thus v ∈ vars e using assms
  by (auto simp add : vars-def)
     (rule-tac ?x=λ v. 0 in exI, rule-tac ?x=1 in exI, simp)+
qed


lemma vars-example-2 :
  fixes e::('v,integer) aexp
  assumes e = (λ σ. σ x − σ y)
  assumes x = y
  shows   vars e = {}
using assms by (auto simp add : vars-def)
```

lemma *vars-example-3* :
fixes e::($'v$,*integer*) *aexp*
assumes $e = (\lambda\ \sigma.\ 0 * \sigma\ x)$
shows *vars* $e = \{\}$
using *assms* **by** (*simp add* : *vars-def*)

A.2.5 Fresh variables

Our notion of symbolic execution suppose *static single assignment form.* In order to symbolically execute an assignment, we require the existence of a fresh symbolic variable for the configuration from which symbolic execution is performed. We define here the notion of *freshness* of a variable for an arithmetic expression.

A variable is fresh for an expression if does not belong to its set of variables.

abbreviation *fresh* ::
$'v \Rightarrow ('v,'d)$ *aexp* $\Rightarrow$ *bool*
where
fresh $v\ e \equiv v \notin$ *vars* e

end
theory *Bexp*
imports *Aexp*
begin

A.3 Boolean Expressions

We proceed as in `Aexp.thy`.

A.3.1 Basic definitions

The *bexp* type-synonym

We represent boolean expressions, their set of variables and the notion of freshness of a variable in the same way than for arithmetic expressions.

type-synonym $('v,'d)$ *bexp* = $('v,'d)$ *state* $\Rightarrow$ *bool*

definition *vars* ::
$('v,'d)$ *bexp* $\Rightarrow$ $'v$ *set*
where
vars $e = \{v.\ \exists\ \sigma\ val.\ e\ (\sigma(v := val)) \neq e\ \sigma\}$

abbreviation *fresh* ::
$'v \Rightarrow ('v,'d)\ bexp \Rightarrow bool$
where
$fresh\ v\ e \equiv v \notin vars\ e$

Satisfiability of an expression

A boolean expression e is satisfiable if there exists a state σ such that $e\ \sigma$ is *true*.

definition *sat* ::
$('v,'d)\ bexp \Rightarrow bool$
where
$sat\ e = (\exists\ \sigma.\ e\ \sigma)$

Entailment

A boolean expression φ entails another boolean expression ψ if all states making φ true also make ψ true.

definition *entails* ::
$('v,'d)\ bexp \Rightarrow ('v,'d)\ bexp \Rightarrow bool$ (**infix** $\models_B$ *55*)
where
$\varphi \models_B \psi \equiv (\forall\ \sigma.\ \varphi\ \sigma \longrightarrow \psi\ \sigma)$

Conjunction

In the following, path predicates are represented by sets of boolean expressions. We define the conjunction of a set of boolean expressions E as the expression that associates *true* to a state σ if, for all elements e of E, e associates *true* to σ.

definition *conjunct* ::
$('v,'d)\ bexp\ set \Rightarrow ('v,'d)\ bexp$
where
$conjunct\ E \equiv (\lambda\ \sigma.\ \forall\ e \in E.\ e\ \sigma)$

A.3.2 Properties about the variables of an expression

As said earlier, our definition of symbolic execution requires the existence of a fresh symbolic variable in the case of an assignment. In the following, a number of proof relies on this fact. We will show the existence of such variables assuming the set of symbolic variables already in use is finite and show that symbolic execution preserves the finiteness of this set, under certain conditions. This in turn requires a number of lemmas about the finiteness of boolean expressions. More precisely, when symbolic execution goes through a guard or an assignment, it conjuncts a new expression to the path predicate. In the case of an assignment, this new expression is an equality linking the new symbolic variable associated to the defined program variable to its symbolic value. In the following, we prove that:

1. the conjunction of a finite set of expressions whose sets of variables are finite has a finite set of variables,
2. the equality of two arithmetic expressions whose sets of variables are finite has a finite set of variables.

Variables of a conjunction

The set of variables of the conjunction of two expressions is a subset of the union of the sets of variables of the two sub-expressions. As a consequence, the set of variables of the conjunction of a finite set of expressions whose sets of variables are finite is also finite.

```
lemma vars-of-conj :
  vars (λ σ. e₁ σ ∧ e₂ σ) ⊆ vars e₁ ∪ vars e₂ (is vars ?e ⊆ vars e₁ ∪ vars e₂)
unfolding subset-iff
proof (intro allI impI)
  fix v assume v ∈ vars ?e

  then obtain σ val
  where ?e (σ (v := val)) ≠ ?e σ
  unfolding vars-def by blast

  hence e₁ (σ (v := val)) ≠ e₁ σ ∨ e₂ (σ (v := val)) ≠ e₂ σ by auto

  thus v ∈ vars e₁ ∪ vars e₂ unfolding vars-def by blast
qed


lemma finite-conj :
  assumes finite E
  assumes ∀ e ∈ E. finite (vars e)
  shows  finite (vars (conjunct E))
using assms
proof (induct rule : finite-induct, goal-cases)
  case 1 thus ?case by (simp add : vars-def conjunct-def)
next
  case (2 e E)

  thus ?case
  using vars-of-conj[of e conjunct E]
  by (rule-tac rev-finite-subset, auto simp add : conjunct-def)
qed
```

Variables of an equality

We proceed analogously for the equality of two arithmetic expressions.

```
lemma vars-of-eq-a :
```

shows $vars\ (\lambda\ \sigma.\ e_1\ \sigma = e_2\ \sigma) \subseteq Aexp.vars\ e_1 \cup Aexp.vars\ e_2$
(**is** $vars\ ?e \subseteq Aexp.vars\ e_1 \cup Aexp.vars\ e_2$)
unfolding *subset-iff*
proof (*intro allI impI*)
fix v **assume** $v \in vars\ ?e$

then obtain σ *val* **where** $?e\ (\sigma\ (v := val)) \neq ?e\ \sigma$ **unfolding** *vars-def* **by** *blast*

hence $e_1\ (\sigma\ (v := val)) \neq e_1\ \sigma \vee e_2\ (\sigma\ (v := val)) \neq e_2\ \sigma$ **by** *auto*

thus $v \in Aexp.vars\ e_1 \cup Aexp.vars\ e_2$ **unfolding** *Aexp.vars-def* **by** *blast*
qed

lemma *finite-vars-of-a-eq* :
assumes *finite* ($Aexp.vars\ e_1$)
assumes *finite* ($Aexp.vars\ e_2$)
shows *finite* ($vars\ (\lambda\ \sigma.\ e_1\ \sigma = e_2\ \sigma)$)
using *assms vars-of-eq-a*[*of* $e_1\ e_2$] **by** (*rule-tac rev-finite-subset*, *auto*)

end
theory *Store*
imports *Aexp Bexp*
begin

A.4 Stores

In this section, we introduce the type of stores, which we use to link program variables with their symbolic counterpart during symbolic execution. We define the notion of consistency of a pair of program and symbolic states w.r.t. a store. This notion will prove helpful when defining various concepts and proving facts related to subsumption (see `Conf.thy`). Finally, we model substitutions that will be performed during symbolic execution (see `SymExec.thy`) by two operations: *adapt-aexp* and *adapt-bexp*.

A.4.1 Basic definitions

The *store* type-synonym

Symbolic execution performs over configurations (see `Conf.thy`), which are pairs made of:

- a *store* mapping program variables to symbolic variables,
- a set of boolean expressions which records constraints over symbolic variables and whose conjunction is the actual path predicate of the configuration.

We define stores as total functions from program variables to indexes.

type-synonym $'v\ store = 'v \Rightarrow nat$

Symbolic variables of a store

The symbolic variable associated to a program variable v by a store s is the couple $(v,\ s\ v)$.

definition *symvar* ::
$'v \Rightarrow 'v\ store \Rightarrow 'v\ symvar$
where
$symvar\ v\ s \equiv (v, s\ v)$

The function associating symbolic variables to program variables obtained from s is injective.

lemma
$inj\ (\lambda\ v.\ symvar\ v\ s)$
by $(auto\ simp\ add : inj\text{-}on\text{-}def\ symvar\text{-}def)$

The sets of symbolic variables of a store is the image set of the function *symvar*.

definition *symvars* ::
$'v\ store \Rightarrow 'v\ symvar\ set$
where
$symvars\ s = (\lambda\ v.\ symvar\ v\ s)\ `\ (UNIV::'v\ set)$

Fresh symbolic variables

A symbolic variable is said to be fresh for a store if it is not a member of its set of symbolic variables.

definition *fresh-symvar* ::
$'v\ symvar \Rightarrow 'v\ store \Rightarrow bool$
where
$fresh\text{-}symvar\ sv\ s = (sv \notin symvars\ s)$

A.4.2 Consistency

We say that a program state σ and a symbolic state σ_{sym} are *consistent* with respect to a store s if, for each variable v, the value associated by σ to v is equal to the value associated by σ_{sym} to the symbolic variable associated to v by s.

definition *consistent* ::
$('v,'d)\ state \Rightarrow ('v\ symvar,'d)\ state \Rightarrow 'v\ store \Rightarrow bool$
where
$consistent\ \sigma\ \sigma_{sym}\ s \equiv (\forall\ v.\ \sigma_{sym}\ (symvar\ v\ s) = \sigma\ v)$

There always exists a couple of consistent states for a given store.

lemma

$\exists\ \sigma\ \sigma_{sym}.$ *consistent* $\sigma\ \sigma_{sym}$ *s*
by (*auto simp add* : *consistent-def*)

Moreover, given a store and a program (resp. symbolic) state, one can always build a symbolic (resp. program) state such that the two states are coherent w.r.t. the store. The four following lemmas show how to build the second state given the first one.

lemma *consistent-eq1* :
consistent $\sigma\ \sigma_{sym}$ $s = (\forall\ sv \in symvars\ s.\ \sigma_{sym}\ sv = \sigma\ (fst\ sv))$
by (*auto simp add* : *consistent-def symvars-def symvar-def*)

lemma *consistent-eq2* :
consistent $\sigma\ \sigma_{sym}$ *store* $= (\sigma = (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ store)))$
by (*auto simp add* : *consistent-def*)

lemma *consistentI1* :
consistent $\sigma\ (\lambda\ sv.\ \sigma\ (fst\ sv))$ *store*
using *consistent-eq1* **by** *fast*

lemma *consistentI2* :
consistent $(\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ store))\ \sigma_{sym}$ *store*
using *consistent-eq2* **by** *fast*

A.4.3 Adaptation of an arithmetic expression to a store

If we were to represent expressions syntactically, as is often the case, then we would need to perform, during symbolic execution of an assignment of an expression e to a variable v, a flat substitution of the program variables occurring in e by their symbolic counterparts given by the current store. Since we model expressions by total functions, we cannot talk about (flat) substitution: we define an equivalent operation that we call the adaptation of the (arithmetic expression) e to a store s.

definition *adapt-aexp* ::
$('v,'d)$ *aexp* $\Rightarrow$ $'v$ *store* $\Rightarrow$ $('v\ symvar,'d)$ *aexp*
where
adapt-aexp e s $= (\lambda\ \sigma_{sym}.\ e\ (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ s)))$

Given an arithmetic expression e, a program state σ and a symbolic state σ_{sym} coherent with a store s, the value associated to σ_{sym} by the adaptation of e to s is the same than the value associated by e to σ. This confirms the fact that *adapt-aexp* models the act of substituting occurrences of program variables by their symbolic counterparts in a term over program variables.

lemma *adapt-aexp-is-subst* :
assumes *consistent* $\sigma\ \sigma_{sym}$ *s*

shows $(adapt\text{-}aexp\ e\ s)\ \sigma_{sym} = e\ \sigma$
using *assms* **by** (*simp add* : *consistent-eq2 adapt-aexp-def*)

As said earlier, we will later need to prove that symbolic execution preserves finiteness of the set of symbolic variables in use, which requires that the adaptation of an arithmetic expression to a store preserves finiteness of the set of variables of expressions. We proceed as follows.

First, we show that if v is a variable of an expression e, then the symbolic variable associated to v by a store is a variable of the adaptation of e to this store.

lemma *var-imp-symvar-var* :
assumes $v \in Aexp.vars\ e$
shows $symvar\ v\ s \in Aexp.vars\ (adapt\text{-}aexp\ e\ s)$ (**is** $?sv \in Aexp.vars\ ?e'$)
proof −
obtain σ *val* **where** $e\ (\sigma\ (v := val)) \neq e\ \sigma$
using *assms* **unfolding** *Aexp.vars-def* **by** *blast*

moreover
have $(\lambda va.\ ((\lambda sv.\ \sigma\ (fst\ sv))(?sv := val))\ (symvar\ va\ s)) = (\sigma(v := val))$
by (*auto simp add* : *symvar-def*)

ultimately
show *?thesis*
unfolding *Aexp.vars-def mem-Collect-eq*
using *consistentI1*[*of* σ *s*]
consistentI2[*of* $(\lambda sv.\ \sigma\ (fst\ sv))(?sv{:=}\ val)$ *s*]
by (*rule-tac* $?x{=}\lambda sv.\ \sigma\ (fst\ sv)$ **in** *exI*, *rule-tac* $?x{=}val$ **in** *exI*)
(*simp add* : *adapt-aexp-is-subst*)
qed

On the other hand, if *sv* is a symbolic variable in the adaptation of an expression to a store, then the program variable it represents is a variable of this expression. This requires to prove that the set of variables of the adaptation of an expression to a store is a subset of the symbolic variables of this store.

lemma *symvars-of-adapt-aexp* :
$Aexp.vars\ (adapt\text{-}aexp\ e\ s) \subseteq symvars\ s$ (**is** $Aexp.vars\ ?e' \subseteq symvars\ s$)
unfolding *subset-iff*
proof (*intro allI impI*)
fix *sv*

assume $sv \in Aexp.vars\ ?e'$

then obtain σ_{sym} *val*
where $?e'\ (\sigma_{sym}\ (sv := val)) \neq\ ?e'\ \sigma_{sym}$
by (*simp add* : *Aexp.vars-def*, *blast*)

hence $(\lambda\ x.\ (\sigma_{sym}\ (sv := val))\ (symvar\ x\ s)) \neq (\lambda\ x.\ \sigma_{sym}\ (symvar\ x\ s))$

```
  proof (intro notI)
    assume (λx. (σ_sym(sv := val)) (symvar x s)) = (λx. σ_sym (symvar x s))

    hence ?e' (σ_sym (sv := val)) = ?e' σ_sym
    by (simp add : adapt-aexp-def)

    thus False
    using ‹?e' (σ_sym (sv := val)) ≠ ?e' σ_sym›
    by (elim notE)
  qed

  then obtain v
  where (σ_sym (sv := val)) (symvar v s) ≠ σ_sym (symvar v s)
  by blast

  hence sv = symvar v s by (case-tac sv = symvar v s, simp-all)

  thus sv ∈ symvars s by (simp add : symvars-def)
qed


lemma symvar-var-imp-var :
  assumes sv ∈ Aexp.vars (adapt-aexp e s) (is sv ∈ Aexp.vars ?e')
  shows   fst sv ∈ Aexp.vars e
proof −
  obtain v where sv = (v, s v)
  using assms(1) symvars-of-adapt-aexp
  unfolding symvars-def symvar-def
  by blast

  obtain σ_sym val where ?e' (σ_sym (sv := val)) ≠ ?e' σ_sym
  using assms unfolding Aexp.vars-def by blast

  moreover
  have (λv. (σ_sym (sv := val))(symvar v s)) = (λv. σ_sym (symvar v s))(v := val)
  using ‹sv = (v, s v)› by (auto simp add : symvar-def)

  ultimately
  show ?thesis
  using ‹sv = (v, s v)›
        consistentI2[of σ_sym s]
        consistentI2[of σ_sym (sv := val) s]
  unfolding Aexp.vars-def
  by (simp add : adapt-aexp-is-subst) blast
qed
```

Thus, we have that the set of variables of the adaptation of an expression to a store is the set of symbolic variables associated by this store to the variables of this expression.

lemma *adapt-aexp-vars* :
Aexp.vars (*adapt-aexp e s*) = (λ *v. symvar v s*) ‘ *Aexp.vars e*
unfolding *set-eq-iff image-def mem-Collect-eq Bex-def*
proof (*intro allI iffI*, *goal-cases*)
case (*1 sv*)

moreover
have *sv* = *symvar* (*fst sv*) *s*
using *1 symvars-of-adapt-aexp*
by (*force simp add*: *symvar-def symvars-def*)

ultimately
show *?case* **using** *symvar-var-imp-var* **by** *blast*
next
case (*2 sv*) **thus** *?case* **using** *var-imp-symvar-var* **by** *fast*
qed

The fact that the adaptation of an arithmetic expression to a store preserves finiteness of the set of variables trivially follows the previous lemma.

lemma *finite-vars-imp-finite-adapt-a* :
assumes *finite* (*Aexp.vars e*)
shows *finite* (*Aexp.vars* (*adapt-aexp e s*))
unfolding *adapt-aexp-vars* **using** *assms* **by** *auto*

A.4.4 Adaptation of a boolean expression to a store

We proceed analogously for the adaptation of boolean expressions to a store.

definition *adapt-bexp* ::
$('v,'d)$ *bexp* $\Rightarrow$ *'v store* $\Rightarrow$ $('v$ *symvar*$,'d)$ *bexp*
where
adapt-bexp e s = (λ σ. *e* (λ *x*. σ (*symvar x s*)))

lemma *adapt-bexp-is-subst* :
assumes *consistent* σ σ_{sym} *s*
shows (*adapt-bexp e s*) σ_{sym} = *e* σ
using *assms* **by** (*simp add* : *consistent-eq2 adapt-bexp-def*)

lemma *var-imp-symvar-var2* :
assumes *v* $\in$ *Bexp.vars e*
shows *symvar v s* $\in$ *Bexp.vars* (*adapt-bexp e s*) (**is** *?sv* $\in$ *Bexp.vars ?e'*)
proof −
obtain σ *val* **where** *A* : *e* (σ (*v* := *val*)) $\neq$ *e* σ
using *assms* **unfolding** *Bexp.vars-def* **by** *blast*

moreover
have (λ*va*. ((λ*sv*. σ (*fst sv*))(*?sv* := *val*)) (*symvar va s*)) = (σ(*v* := *val*))
by (*auto simp add* : *symvar-def*)

ultimately
show *?thesis*
unfolding *Bexp.vars-def mem-Collect-eq*
using *consistentI1*[*of* σ *s*]
consistentI2[*of* (λ*sv*. σ (*fst sv*))(*?sv*:= *val*) *s*]
by (*rule-tac ?x*=λ*sv*. σ (*fst sv*) **in** *exI*, *rule-tac ?x*=*val* **in** *exI*)
(*simp add* : *adapt-bexp-is-subst*)
qed

lemma *symvars-of-adapt-bexp* :
Bexp.vars (*adapt-bexp e s*) $\subseteq$ *symvars s* (**is** *Bexp.vars ?e'* $\subseteq$ *?SV*)
proof
fix *sv*
assume *sv* $\in$ *Bexp.vars ?e'*

then obtain σ_{sym} *val*
where *?e'* (σ_{sym} (*sv* := *val*)) $\neq$ *?e'* σ_{sym}
by (*simp add* : *Bexp.vars-def*, *blast*)

hence (λ *x*. (σ_{sym} (*sv* := *val*)) (*symvar x s*)) $\neq$ (λ *x*. σ_{sym} (*symvar x s*))
by (*auto simp add* : *adapt-bexp-def*)

hence $\exists$ *v*. (σ_{sym} (*sv* := *val*)) (*symvar v s*) $\neq$ σ_{sym} (*symvar v s*) **by** *force*

then obtain *v*
where (σ_{sym} (*sv* := *val*)) (*symvar v s*) $\neq$ σ_{sym} (*symvar v s*)
by *blast*

hence *sv* = *symvar v s* **by** (*case-tac sv* = *symvar v s*, *simp-all*)

thus *sv* $\in$ *symvars s* **by** (*simp add* : *symvars-def*)
qed

lemma *symvar-var-imp-var2* :
assumes *sv* $\in$ *Bexp.vars* (*adapt-bexp e s*) (**is** *sv* $\in$ *Bexp.vars ?e'*)
shows *fst sv* $\in$ *Bexp.vars e*
proof −
obtain *v* **where** *sv* = (*v*, *s v*)
using *assms symvars-of-adapt-bexp*
unfolding *symvars-def symvar-def*
by *blast*

obtain σ_{sym} *val* **where** *?e'* (σ_{sym} (*sv* := *val*)) $\neq$ *?e'* σ_{sym}
using *assms* **unfolding** *vars-def* **by** *blast*

moreover

have $(\lambda v.\ (\sigma_{sym}\ (sv := val))(symvar\ v\ s)) = (\lambda v.\ \sigma_{sym}\ (symvar\ v\ s))(v := val)$
using ‹*sv* = (*v*, *s v*)› **by** (*auto simp add* : *symvar-def*)

ultimately
show *?thesis*
using ‹*sv* = (*v*, *s v*)›
consistentI2[*of* σ_{sym} *s*]
consistentI2[*of* σ_{sym} (*sv* := *val*) *s*]
unfolding *vars-def* **by** (*simp add* : *adapt-bexp-is-subst*) *blast*
qed

lemma *adapt-bexp-vars* :
Bexp.vars (*adapt-bexp e s*) = (λ *v*. *symvar v s*) ‘ *Bexp.vars e*
(**is** *Bexp.vars ?e′* = *?R*)
unfolding *set-eq-iff image-def mem-Collect-eq Bex-def*
proof (*intro allI iffI*, *goal-cases*)
case (*1 sv*)

hence *fst sv* ∈ *vars e* **by** (*rule symvar-var-imp-var2*)

moreover
have *sv* = *symvar* (*fst sv*) *s*
using *1 symvars-of-adapt-bexp*
by (*force simp add*: *symvar-def symvars-def*)

ultimately
show *?case* **by** *blast*
next
case (*2 sv*)

then obtain *v* **where** *v* ∈ *vars e sv* = *symvar v s* **by** *blast*

thus *?case* **using** *var-imp-symvar-var2* **by** *simp*
qed

lemma *finite-vars-imp-finite-adapt-b* :
assumes *finite* (*Bexp.vars e*)
shows *finite* (*Bexp.vars* (*adapt-bexp e s*))
unfolding *adapt-bexp-vars* **using** *assms* **by** *auto*

end
theory *Conf*
imports *Store Finite-Set*
begin

A.5 Configurations and Subsumption

In this section, we first introduce most elements related to our modeling of program behaviors. We first define the type of configurations, on which symbolic execution performs, and define the various concepts we will rely upon in the following and state and prove properties about them.

A.5.1 Configurations

Configurations are pairs (*store*, *pred*) where:

- *store* is a store mapping program variable to symbolic variables,
- *pred* is a set of boolean expressions over program variables whose conjunction is the actual path predicate.

record $('v,'d)$ *conf* $=$
store $::$ $'v$ *store*
pred $::$ $('v$ *symvar*$,'d)$ *bexp set*

A.5.2 Symbolic variables of a configuration.

The set of symbolic variables of a configuration is the union of the set of symbolic variables of its store component with the set of variables of its path predicate.

definition *symvars* $::$
$('v,'d)$ *conf* $\Rightarrow$ $'v$ *symvar set*
where
symvars c $=$ *Store.symvars* (*store c*) $\cup$ *Bexp.vars* (*conjunct* (*pred c*))

A.5.3 Freshness.

A symbolic variable is said to be fresh for a configuration if it is not an element of its set of symbolic variables.

definition *fresh-symvar* $::$
$'v$ *symvar* $\Rightarrow$ $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
fresh-symvar sv c $=$ (*sv* $\notin$ *symvars c*)

A.5.4 Satisfiability

A configuration is said to be satisfiable if its path predicate is satisfiable.

abbreviation *sat* $::$
$('v,'d)$ *conf* $\Rightarrow$ *bool*
where
sat c $\equiv$ *Bexp.sat* (*conjunct* (*pred c*))

A.5.5 States of a configuration

Configurations represent sets of program states. The set of program states represented by a configuration, or simply its set of program states, is defined as the set of program states such that consistent symbolic states w.r.t. the store component of the configuration satisfies its path predicate.

definition *states* ::
  $('v,'d)\ conf \Rightarrow ('v,'d)\ state\ set$
where
 $states\ c = \{\sigma.\ \exists\ \sigma_{sym}.\ consistent\ \sigma\ \sigma_{sym}\ (store\ c) \wedge conjunct\ (pred\ c)\ \sigma_{sym}\}$

A configuration is satisfiable if and only if its set of states is not empty.

lemma *sat-eq* :
  $sat\ c = (states\ c \neq \{\})$
using *consistentI2* **by** (*simp add* : *sat-def states-def*) *fast*

A.5.6 Subsumption

A configuration c_2 is subsumed by a configuration c_1 if the set of states of c_2 is a subset of the set of states of c_1.

definition *subsums* ::
  $('v,'d)\ conf \Rightarrow ('v,'d)\ conf \Rightarrow bool$ (**infixl** $\sqsubseteq$ *55*)
where
  $c_2 \sqsubseteq c_1 \equiv (states\ c_2 \subseteq states\ c_1)$

The subsumption relation is reflexive and transitive.

lemma *subsums-refl* :
  $c \sqsubseteq c$
by (*simp only* : *subsums-def*)

lemma *subsums-trans* :
  $c_1 \sqsubseteq c_2 \Longrightarrow c_2 \sqsubseteq c_3 \Longrightarrow c_1 \sqsubseteq c_3$
unfolding *subsums-def* **by** *simp*

However, it is not anti-symmetric. This is due to the fact that different configurations can have the same sets of program states. However, the following lemma trivially follows the definition of subsumption.

lemma
  assumes $c_1 \sqsubseteq c_2$
  assumes $c_2 \sqsubseteq c_1$
  shows $states\ c_1 = states\ c_2$
using *assms* **by** (*simp add* : *subsums-def*)

A satisfiable configuration can only be subsumed by satisfiable configurations.

lemma *sat-sub-by-sat* :
  assumes $sat\ c_2$

and $c_2 \sqsubseteq c_1$
shows *sat* c_1
using *assms sat-eq*[*of* c_1] *sat-eq*[*of* c_2]
by (*simp add* : *subsums-def*) *fast*

On the other hand, an unsatisfiable configuration can only subsume unsatisfiable configurations.

lemma *unsat-subs-unsat* :
assumes $\neg$ *sat* c_1
assumes $c_2 \sqsubseteq c_1$
shows $\neg$ *sat* c_2
using *assms sat-eq*[*of* c_1] *sat-eq*[*of* c_2]
by (*simp add* : *subsums-def*)

A.5.7 Semantics of a configuration

The semantics of a configuration c is a boolean expression e over program states associating *true* to a program state if it is a state of c. In practice, given two configurations c_1 and c_2, it is not possible to enumerate their sets of states to establish the inclusion in order to detect a subsumption. We detect the subsumption of the former by the latter by asking a constraint solver if *sem* c_1 entails *sem* c_2. The following theorem shows that the way we detect subsumption in practice is correct.

definition *sem* ::
$('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *bexp*
where
sem $c = (\lambda\ \sigma.\ \sigma \in$ *states* $c)$

theorem *subsum-eq-sem-entailment* :
$c_2 \sqsubseteq c_1 \longleftrightarrow$ *sem* $c_2 \models_B$ *sem* c_1
unfolding *subsums-def sem-def subset-iff entails-def* **by** (*rule refl*)

A.5.8 Entailment

A configuration *entails* a boolean expression if its semantics entails this expression. This is equivalent to say that this expression holds for any state of this configuration.

abbreviation *entails* ::
$('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *bexp* $\Rightarrow$ *bool* (**infixl** $\models_c$ *55*)
where
$c \models_c \varphi \equiv$ *sem* $c \models_B \varphi$

lemma
sem $c \models_B e \longleftrightarrow (\forall\ \sigma \in$ *states* $c.\ e\ \sigma)$
by (*auto simp add* : *states-def sem-def entails-def*)

A.5.9 Abstractions

Abstracting a configuration consists in removing a given expression from its *pred* component, i.e. weakening its path predicate. This definition of abstraction motivates the fact that the *pred* component of configurations has been defined as a set of boolean expressions instead of a boolean expression.

definition *abstract* ::
  $('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
  abstract $c\ c_a \equiv c \sqsubseteq c_a$

end
theory *Label*
imports *Aexp Bexp*
begin

A.6 Symbolic Execution

In this section, we introduce our notion of symbolic execution. After introducing labels and defining symbolic execution, we give a number of basic properties about the latter. One of our main objective here is to prove that symbolic execution is monotonic with respect to the subsumption relation, which is a crucial point in order to prove the main theorems of `RB.thy`. Moreover, Isabelle/HOL requires the actual formalization of a number of facts one would not worry when implementing or writing a pen-and-paper proof. Here, we will need to prove that there exist successors of the configurations on which symbolic execution is performed. Although this seems quite obvious in practice, proofs of such facts will be needed a number of times in the following theories. Finally, we define the feasibility of a sequence of labels.

A.6.1 Labels

In the following, we model programs by control flow graphs where edges (rather than vertices) are labeled with either assignments or with the condition associated with a branch of a conditional statement. We put a label on every edge : statements that do not modify the program state (like `jump`, `break`, etc) are labeled by a *Skip*.

datatype $('v,'d)$ *label* = *Skip* | *Assume* $('v,'d)$ *bexp* | *Assign* $'v$ $('v,'d)$ *aexp*

We say that a label is *finite* if the set of variables of its sub-expression is finite (*Skip* labels are thus considered finite).

definition *finite-label* ::
  $('v,'d)$ *label* $\Rightarrow$ *bool*
where

```
finite-label l ≡ case l of
                   Assume e   ⇒ finite (Bexp.vars e)
                 | Assign - e ⇒ finite (Aexp.vars e)
                 | -          ⇒ True
```

```
abbreviation finite-labels ::
  ('v,'d) label list ⇒ bool
where
  finite-labels ls ≡ (∀ l ∈ set ls. finite-label l)

end
theory SymExec
imports Finite-Set Label Conf
begin
```

A.6.2 Definitions of *SE* and *SE_star*

We model symbolic execution by an inductive predicate *SE* that takes two configurations c_1 and c_2 and a label l and evaluates to *true* if and only if c_2 is a *possible result* of the symbolic execution of l from c_1. We say that c_2 is a possible result because, when l is of the form *Assign v e*, we associate a fresh symbolic variable to the program variable v, but we do no specify how this fresh variable is chosen (see the two assumptions in the third case). We could have model *SE* (and *SE-star*) by a function producing the new configuration, instead of using inductive predicates. However this would require to provide the two said assumptions in each lemma involving *SE*, which is not necessary using a predicate. Modeling symbolic execution in this way has the advantage that it simplifies the following proofs while not requiring additional lemmas.

Symbolic execution of *Skip* does not change the configuration from which it is performed.

When the label is of the form *Assume e*, the adaptation of e to the store is added to the *pred* component.

In the case of an assignment, the *store* component is updated such that it now maps a fresh symbolic variable to the assigned program variable. A constraint relating this program variable with its new symbolic value is added to the *pred* component.

The second assumption in the third case requires that there exists at least one fresh symbolic variable for *c*. In the following, a number of theorems are needed to show that such variables exist for the configurations on which symbolic execution is performed.

```
inductive SE ::
  ('v,'d) conf ⇒ ('v,'d) label ⇒ ('v,'d) conf ⇒ bool
```

where
SE c Skip c

| *SE c (Assume e)* ⦇ *store = store c, pred = pred c* ∪ {*adapt-bexp e (store c)*} ⦈

| *fst sv = v* ⟹
fresh-symvar sv c ⟹
SE c (Assign v e) ⦇ *store = (store c)(v := snd sv),*
pred = pred c ∪ {(λ σ. σ *sv* = (*adapt-aexp e (store c)*) σ)} ⦈

lemma
assumes *SE* c_1 *(Assign v e)* c_2
shows ∃ *sv. fst sv = v* ∧ *fresh-symvar sv* c_1
using *assms* **by** (*simp add* : *SE.simps*) *blast*

In the same spirit, we extend symbolic execution to sequence of labels.

inductive *SE-star* ::
('v,'d) conf ⇒ *('v,'d) label list* ⇒ *('v,'d) conf* ⇒ *bool*
where
SE-star c [] *c*
| *SE* c_1 *l* c_2 ⟹ *SE-star* c_2 *ls* c_3 ⟹ *SE-star* c_1 (*l* # *ls*) c_3

A.6.3 Basic properties of *SE*

If symbolic execution yields a satisfiable configuration, then it has been performed from a satisfiable configuration.

lemma *SE-sat-imp-sat* :
assumes *SE c l c'*
assumes *sat c'*
shows *sat c*
using *assms* **by** *cases* (*auto simp add* : *sat-def conjunct-def*)

If symbolic execution is performed from an unsatisfiable configuration, then it will yield an unsatisfiable configuration.

lemma *unsat-imp-SE-unsat* :
assumes *SE c l c'*
assumes ¬ *sat c*
shows ¬ *sat c'*
using *assms* **by** *cases* (*simp add* : *sat-def conjunct-def*)+

Given two configurations c and c' and a label l such that *SE c l c'*, the three following lemmas express c' as a function of c.

lemma [*simp*] :
SE c Skip c' = (*c'* = *c*)
by (*simp add* : *SE.simps*)

lemma *SE-Assume-eq* :
  SE c (*Assume e*) c' =
   (c' = (| *store* = *store c*, *pred* = *pred c* ∪ {*adapt-bexp e* (*store c*)} |))
by (*simp add* : *SE.simps*)

lemma *SE-Assign-eq* :
  SE c (*Assign v e*) c' =
  (∃ *sv*. *fresh-symvar sv c*
     ∧ *fst sv* = *v*
     ∧ c' = (| *store* = (*store c*)(*v* := *snd sv*),
          pred = *insert* (λσ. σ *sv* = *adapt-aexp e* (*store c*) σ) (*pred c*)|))
by (*simp only* : *SE.simps*, *blast*)

Given two configurations c and c' and a label l such that *SE* $c\ l\ c'$, the two following lemmas express the path predicate of c' as a function of the path predicate of c when l models a guard or an assignment.

lemma *path-pred-of-SE-Assume* :
  assumes *SE c* (*Assume e*) c'
  shows *conjunct* (*pred* c') =
       (λ σ. *conjunct* (*pred c*) σ ∧ *adapt-bexp e* (*store c*) σ)
using *assms SE-Assume-eq*[*of c e c'*]
by (*auto simp add* : *conjunct-def*)

lemma *path-pred-of-SE-Assign* :
  assumes *SE c* (*Assign v e*) c'
  shows ∃ *sv*. *conjunct* (*pred* c') =
       (λ σ. *conjunct* (*pred c*) σ ∧ σ *sv* = *adapt-aexp e* (*store c*) σ)
using *assms SE-Assign-eq*[*of c v e c'*]
by (*fastforce simp add* : *conjunct-def*)

Let c and c' be two configurations such that c' is obtained from c by symbolic execution of a label of the form *Assume e*. The states of c' are the states of c that satisfy e. This theorem will help prove that symbolic execution is monotonic w.r.t. subsumption.

theorem *states-of-SE-assume* :
  assumes *SE c* (*Assume e*) c'
  shows *states* c' = {σ ∈ *states c*. *e* σ}
using *assms SE-Assume-eq*[*of c e c'*]
by (*auto simp add* : *adapt-bexp-is-subst states-def conjunct-def*)

Let c and c' be two configurations such that c' is obtained from c by symbolic execution of a label of the form *Assign v e*. We want to express the set of states of c' as a function of the set of states of c. Since the proof requires a number of details, we split it into two sub lemmas.

First, we show that if σ' is a state of c', then it has been obtain from an adequate update of a state σ of c.

lemma *states-of-SE-assign1* :
  assumes *SE c (Assign v e) c'*
  assumes $\sigma' \in$ *states c'*
  shows $\exists\ \sigma \in$ *states c*. $\sigma' = (\sigma\ (v := e\ \sigma))$
proof −
  obtain σ_{sym}
  where *1* : *consistent* $\sigma'\ \sigma_{sym}$ (*store c'*)
  and *2* : *conjunct* (*pred c'*) σ_{sym}
  using *assms*(*2*) **unfolding** *states-def* **by** *blast*

  then obtain σ
  where *3* : *consistent* $\sigma\ \sigma_{sym}$ (*store c*)
  using *consistentI2* **by** *blast*

  moreover
  have *conjunct* (*pred c*) σ_{sym}
  using *assms*(*1*) *2* **by** (*auto simp add* : *SE-Assign-eq conjunct-def*)

  ultimately
  have $\sigma \in$ *states c* **by** (*simp add* : *states-def*) *blast*

  moreover
  have $\sigma' = \sigma\ (v := e\ \sigma)$
  proof −
    have $\sigma'\ v = e\ \sigma$
    proof −
      have $\sigma'\ v = \sigma_{sym}$ (*symvar v* (*store c'*))
      using *1* **by** (*simp add* : *consistent-def*)

      moreover
      have σ_{sym} (*symvar v* (*store c'*)) = (*adapt-aexp e* (*store c*)) σ_{sym}
      using *assms*(*1*) *2 SE-Assign-eq*[*of c v e c'*]
      by (*force simp add* : *symvar-def conjunct-def*)

      moreover
      have (*adapt-aexp e* (*store c*)) $\sigma_{sym} = e\ \sigma$
      using *3* **by** (*rule adapt-aexp-is-subst*)

      ultimately
      show *?thesis* **by** *simp*
    qed

    moreover
    have $\forall\ x.\ x \neq v \longrightarrow \sigma'\ x = \sigma\ x$
    proof (*intro allI impI*)
      fix x

      assume $x \neq v$

```
      moreover
      hence σ' x = σ_sym (symvar x (store c))
      using assms(1) 1 unfolding consistent-def symvar-def
      by (drule-tac ?x=x in spec) (auto simp add : SE-Assign-eq)

      moreover
      have σ_sym (symvar x (store c)) = σ x
      using 3 by (auto simp add : consistent-def)

      ultimately
      show σ' x = σ x by simp
    qed

    ultimately
    show ?thesis by auto
  qed

  ultimately
  show ?thesis by (simp add : states-def) blast
qed
```

Then, we show that if there exists a state σ of c from which σ' is obtained by an adequate update, then σ' is a state of c'.

```
lemma states-of-SE-assign2 :
  assumes SE c (Assign v e) c'
  assumes ∃ σ ∈ states c. σ' = σ (v := e σ)
  shows   σ' ∈ states c'
proof −
  obtain σ
  where σ ∈ states c
  and   σ' = σ (v := e σ)
  using assms(2) by blast

  then obtain σ_sym
  where 1 : consistent σ σ_sym (store c)
  and   2 : conjunct (pred c) σ_sym
  unfolding states-def by blast

  obtain sv
  where 3 : fresh-symvar sv c
  and   4 : fst sv = v
  and   5 : c' = (| store = (store c)(v := snd sv),
                    pred   = insert (λσ. σ sv = adapt-aexp e (store c) σ) (pred c) |)
  using assms(1) SE-Assign-eq[of c v e c'] by blast

  def σ_sym' ≡ σ_sym (sv := e σ)

  have consistent σ' σ_sym' (store c')
  using ‹σ' = σ (v := e σ)› 1 4 5
```

```
  by (auto simp add : symvar-def consistent-def σsym'-def)

  moreover
  have conjunct (pred c') σsym'
  proof -
    have conjunct (pred c) σsym'
   using 2 3 by (simp add : fresh-symvar-def symvars-def Bexp.vars-def σsym'-def)

    moreover
    have σsym' sv = (adapt-aexp e (store c)) σsym'
    proof -
      have Aexp.fresh sv (adapt-aexp e (store c))
      using 3 symvars-of-adapt-aexp[of e store c]
      by (auto simp add : fresh-symvar-def symvars-def)

      thus ?thesis
      using adapt-aexp-is-subst[OF 1, of e]
      by (simp add : Aexp.vars-def σsym'-def)
    qed

    ultimately
    show ?thesis using 5 by (simp add: conjunct-def)
  qed

  ultimately
  show ?thesis unfolding states-def by blast
qed
```

The following theorem expressing the set of states of c' as a function of the set of states of c trivially follows the two preceding lemmas.

```
theorem states-of-SE-assign :
  assumes SE c (Assign v e) c'
  shows   states c' = {σ (v := e σ) | σ. σ ∈ states c}
using assms states-of-SE-assign1 states-of-SE-assign2 by fast
```

A.6.4 Monotonicity of *SE*

We are now ready to prove that symbolic execution is monotonic with respect to subsumption.

```
theorem SE-mono-for-sub :
  assumes SE c1 l c1'
  assumes SE c2 l c2'
  assumes c2 ⊑ c1
  shows   c2' ⊑ c1'
using assms
by (cases l,
    (simp,
    (simp add : states-of-SE-assume subsums-def, blast),
    (simp add : states-of-SE-assign subsums-def, blast)))
```

A stronger version of the previous theorem: symbolic execution is monotonic with respect to states equality.

theorem *SE-mono-for-states-eq* :
  assumes *states* c_1 = *states* c_2
  assumes *SE* c_1 *l* c_1'
  assumes *SE* c_2 *l* c_2'
  shows *states* c_2' = *states* c_1'
using *assms*(*1*)
    SE-mono-for-sub[*OF assms*(*2*,*3*)]
    SE-mono-for-sub[*OF assms*(*3*,*2*)]
by (*simp add* : *subsums-def*)

The previous theorem confirms the fact that the way the fresh symbolic variable is chosen in the case of symbolic execution of an assignment does not matter as long as the new symbolic variable is indeed fresh, which is more precisely expressed by the following lemma.

lemma
  assumes *SE c l* c_1
  assumes *SE c l* c_2
  shows *states* c_1 = *states* c_2
using *assms SE-mono-for-states-eq* **by** *fast*

A.6.5 Basic properties of *SE_star*

Some simplification lemmas for *SE-star*.

lemma [*simp*] :
  SE-star c [] c' = (c' = *c*)
by (*subst SE-star.simps*) *auto*

lemma *SE-star-Cons* :
  SE-star c_1 (*l* # *ls*) c_2 = (∃ *c*. *SE* c_1 *l c* ∧ *SE-star c ls* c_2)
by (*subst* (*1*) *SE-star.simps*) *blast*

lemma *SE-star-one* :
  SE-star c_1 [*l*] c_2 = *SE* c_1 *l* c_2
using *SE-star-Cons* **by** *force*

lemma *SE-star-append* :
  SE-star c_1 (ls_1 @ ls_2) c_2 = (∃ *c*. *SE-star* c_1 ls_1 *c* ∧ *SE-star c* ls_2 c_2)
using *assms* **by** (*induct* ls_1 *arbitrary* : c_1, *simp-all add* : *SE-star-Cons*) *blast*

lemma *SE-star-append-one* :
  SE-star c_1 (*ls* @ [*l*]) c_2 = (∃ *c*. *SE-star* c_1 *ls c* ∧ *SE c l* c_2)
unfolding *SE-star-append SE-star-one* **by** (*rule refl*)

Symbolic execution of a sequence of labels from an unsatisfiable configuration yields an unsatisfiable configuration.

lemma *unsat-imp-SE-star-unsat* :
  assumes *SE-star c ls c'*
  assumes ¬ *sat c*
  shows ¬ *sat c'*
using *assms*
by (*induct ls arbitrary* : *c*)
  (*simp*, *force simp add* : *SE-star-Cons unsat-imp-SE-unsat*)

If symbolic execution yields a satisfiable configuration, then it has been performed from a satisfiable configuration.

lemma *SE-star-sat-imp-sat* :
  assumes *SE-star c ls c'*
  assumes *sat c'*
  shows *sat c*
using *assms*
by (*induct ls arbitrary* : *c*)
  (*simp*, *force simp add* : *SE-star-Cons SE-sat-imp-sat*)

A.6.6 Monotonicity of *SE_star*

Monotonicity of *SE* extends to *SE-star*.

theorem *SE-star-mono-for-sub* :
  assumes *SE-star* c_1 *ls* c_1'
  assumes *SE-star* c_2 *ls* c_2'
  assumes $c_2 \sqsubseteq c_1$
  shows $c_2' \sqsubseteq c_1'$
using *assms*
by (*induct ls arbitrary* : c_1 c_2)
  (*auto simp add* : *SE-star-Cons SE-mono-for-sub*)

lemma *SE-star-mono-for-states-eq* :
  assumes *states* c_1 = *states* c_2
  assumes *SE-star* c_1 *ls* c_1'
  assumes *SE-star* c_2 *ls* c_2'
  shows *states* c_2' = *states* c_1'
using *assms*(*1*)
    SE-star-mono-for-sub[*OF assms*(*2*,*3*)]
    SE-star-mono-for-sub[*OF assms*(*3*,*2*)]
by (*simp add* : *subsums-def*)

lemma *SE-star-succs-states* :
  assumes *SE-star c ls* c_1
  assumes *SE-star c ls* c_2
  shows *states* c_1 = *states* c_2
using *assms SE-star-mono-for-states-eq* **by** *fast*

A.6.7 Existence of successors

Here, we are interested in proving that, under certain assumptions, there will always exist fresh symbolic variables for configurations on which symbolic execution is performed. Thus symbolic execution cannot "block" when an assignment is met. For symbolic execution not to block in this case, the configuration from which it is performed must be such that there exist fresh symbolic variables for each program variable. Such configurations are said to be *updatable*.

definition *updatable* ::
 $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
 updatable $c \equiv \forall\ v.\ \exists\ sv.\ fst\ sv = v \wedge$ *fresh-symvar* $sv\ c$

The following lemma shows that being updatable is a sufficient condition for a configuration in order for *SE* not to block.

lemma *updatable-imp-ex-SE-succ* :
 assumes *updatable c*
 shows $\exists\ c'.\ SE\ c\ l\ c'$
using *assms*
by (*cases l*, *simp-all add* : *SE-Assume-eq SE-Assign-eq updatable-def*)

A sufficient condition for a configuration to be updatable is that its path predicate has a finite number of variables. The *store* component has no influence here, since its set of symbolic variables is always a strict subset of the set of symbolic variables (i.e. there always exist fresh symbolic variables for a store). To establish this proof, we need the following intermediate lemma.

We want to prove that if the set of symbolic variables of the path predicate of a configuration is finite, then we can find a fresh symbolic variable for it. However, we express this with a more general lemma. We show that given a finite set of symbolic variables SV and a program variable v such that there exist symbolic variables in SV that are indexed versions of v, then there exists a symbolic variable for v whose index is greater or equal than the index of any other symbolic variable for v in SV.

lemma *finite-symvars-imp-ex-greatest-symvar* :
 fixes SV :: *'a symvar set*
 assumes *finite* SV
 assumes $\exists\ sv \in SV.\ fst\ sv = v$
 shows $\exists\ sv\ \in \{sv \in SV.\ fst\ sv = v\}.$
 $\quad\forall\ sv' \in \{sv \in SV.\ fst\ sv = v\}.\ snd\ sv' \leq snd\ sv$
proof –
 have *finite* $(snd\ `\ \{sv \in SV.\ fst\ sv = v\})$
 and $snd\ `\ \{sv \in SV.\ fst\ sv = v\} \neq \{\}$
 using *assms* **by** *auto*

```
  moreover
  have ∀ (E::nat set). finite E ∧ E ≠ {} ⟶ (∃ n ∈ E. ∀ m ∈ E. m ≤ n)
  by (intro allI impI, induct-tac rule : finite-ne-induct)
     (simp+, force)

  ultimately
  obtain n
  where n ∈ snd ‘ {sv ∈ SV. fst sv = v}
  and   ∀ m ∈ snd ‘ {sv ∈ SV. fst sv = v}. m ≤ n
  by blast

  moreover
  then obtain sv
  where sv ∈ {sv ∈ SV. fst sv = v} and snd sv = n
  by blast

  ultimately
  show ?thesis by blast
qed
```

Thus, a configuration whose path predicate has a finite set of variables is updatable. For example, for any program variable v, the symbolic variable $(v,i+1)$ is fresh for this configuration, where i is the greater index associated to v among the symbolic variables of this configuration. In practice, this is how we choose the fresh symbolic variable.

```
lemma finite-pred-imp-SE-updatable :
  assumes finite (Bexp.vars (conjunct (pred c))) (is finite ?V)
  shows   updatable c
unfolding updatable-def
proof (intro allI)
  fix v

  show ∃ sv. fst sv = v ∧ fresh-symvar sv c
  proof (case-tac ∃ sv ∈ ?V. fst sv = v, goal-cases)
    case 1

    then obtain max-sv
    where       max-sv ∈ ?V
    and         fst max-sv = v
    and   max : ∀ sv'∈{sv ∈ ?V. fst sv = v}. snd sv' ≤ snd max-sv
    using assms finite-symvars-imp-ex-greatest-symvar[of ?V v]
    by blast

    show ?thesis
    using max
    unfolding fresh-symvar-def symvars-def Store.symvars-def symvar-def
    proof (case-tac snd max-sv ≤ store c v, goal-cases)
      case 1 thus ?case by (rule-tac ?x=(v,Suc (store c v)) in exI) auto
    next
```

```
    case 2 thus ?case by (rule-tac ?x=(v,Suc (snd max-sv)) in exI) auto
  qed
 next
  case 2 thus ?thesis
  by (rule-tac ?x=(v, Suc (store c v)) in exI)
    (auto simp add : fresh-symvar-def symvars-def Store.symvars-def symvar-def)
 qed
qed
```

The path predicate of a configuration whose *pred* component is finite and whose elements all have finite sets of variables has a finite set of variables. Thus, this configuration is updatable, and it has a successor by symbolic execution of any label. The following lemma starts from these two assumptions and use the previous ones in order to directly get to the conclusion (this will ease some of the following proofs).

```
lemma finite-imp-ex-SE-succ :
 assumes finite (pred c)
 assumes ∀ e ∈ pred c. finite (Bexp.vars e)
 shows  ∃ c'. SE c l c'
using finite-pred-imp-SE-updatable[OF finite-conj[OF assms(1,2)]]
by (rule updatable-imp-ex-SE-succ)
```

For symbolic execution not to block *along a sequence of labels*, it is not sufficient for the first configuration to be updatable. It must also be such that (all) its successors are updatable. A sufficient condition for this is that the set of variables of its path predicate is finite and that the sub-expression of the label that is executed also has a finite set of variables. Under these assumptions, symbolic execution preserves finiteness of the *pred* component and of the sets of variables of its elements. Thus, successors *SE* are also updatable because they also have a path predicate with a finite set of variables. In the following, to prove this we need two intermediate lemmas:

- one stating that symbolic execution perserves the finiteness of the set of variables of the elements of the *pred* component, provided that the sub expression of the label that is executed has a finite set of variables,
- one stating that symbolic execution preserves the finiteness of the *pred* component.

```
lemma SE-preserves-finiteness1 :
 assumes finite-label l
 assumes SE c l c'
 assumes ∀ e ∈ pred c. finite (Bexp.vars e)
 shows  ∀ e ∈ pred c'. finite (Bexp.vars e)
proof (cases l)
 case Skip thus ?thesis using assms by simp
next
```

```
  case (Assume e) thus ?thesis
  using assms finite-vars-imp-finite-adapt-b
  by (auto simp add : SE-Assume-eq finite-label-def)
next
  case (Assign v e)

  then obtain sv
  where fresh-symvar sv c
  and   fst sv = v
  and   c' = (| store = (store c)(v := snd sv),
              pred  = insert (λσ. σ sv = adapt-aexp e (store c) σ) (pred c)|)
  using assms(2) SE-Assign-eq[of c v e c'] by blast

  moreover
  have finite (Bexp.vars (λσ. σ sv = adapt-aexp e (store c) σ))
  proof -
    have finite (Aexp.vars (λσ. σ sv))
    by (auto simp add : Aexp.vars-def)

    moreover
    have finite (Aexp.vars (adapt-aexp e (store c)))
    using assms(1) Assign finite-vars-imp-finite-adapt-a
    by (auto simp add : finite-label-def)

    ultimately
    show ?thesis using finite-vars-of-a-eq by auto
  qed

  ultimately
  show ?thesis using assms by auto
qed


lemma SE-preserves-finiteness2 :
  assumes SE c l c'
  assumes finite (pred c)
  shows   finite (pred c')
using assms
by (cases l)
   (auto simp add :  SE-Assume-eq SE-Assign-eq)
```

We are now ready to prove that a sufficient condition for symbolic execution not to block along a sequence of labels is that the *pred* component of the "initial configuration" is finite, as well as the set of variables of its elements, and that the sub-expression of the label that is executed also has a finite set of variables.

```
lemma finite-imp-ex-SE-star-succ :
  assumes finite (pred c)
  assumes ∀ e ∈ pred c. finite (Bexp.vars e)
```

```
  assumes finite-labels ls
  shows  ∃ c'. SE-star c ls c'
using assms
proof (induct ls arbitrary : c, goal-cases)
  case 1 show ?case using SE-star.simps by blast
next
  case (2 l ls c)

  then obtain c1 where SE c l c1 using finite-imp-ex-SE-succ by blast

  hence finite (pred c1)
  and  ∀ e ∈ pred c1. finite (Bexp.vars e)
  using 2 SE-preserves-finiteness1 SE-preserves-finiteness2 by fastforce+

  moreover
  have finite-labels ls using 2 by simp

  ultimately
  obtain c2 where SE-star c1 ls c2 using 2 by blast

  thus ?case using ‹SE c l c1› using SE-star-Cons by blast
qed
```

A.6.8 Feasibility of a sequence of labels

A sequence of labels *ls* is said to be feasible from a configuration *c* if there exists a satisfiable configuration *c'* obtained by symbolic execution of *ls* from *c*.

```
definition feasible ::
  ('v,'d) conf ⇒ ('v,'d) label list ⇒ bool
where
  feasible c ls ≡ (∃ c'. SE-star c ls c' ∧ sat c')
```

A simplification lemma for the case where *ls* is not empty.

```
lemma feasible-Cons :
  feasible c (l#ls) = (∃ c'. SE c l c' ∧ sat c' ∧ feasible c' ls)
proof (intro iffI, goal-cases)
  case 1 thus ?case
  using SE-star-sat-imp-sat by (simp add : feasible-def SE-star-Cons) blast
next
  case 2 thus ?case
  using assms unfolding feasible-def SE-star-Cons by blast
qed
```

The following theorem is very important for the rest of this formalization. It states that, given two configurations *c1* and *c2* such that *c1* subsumes *c2*, then any feasible sequence of labels from *c2* is also feasible from *c1*. This is a crucial point in order to prove that our approach preserves the set of feasible paths of the original LTS. This proof requires a number of assumptions

about the finiteness of the sequence of labels, of the path predicates of the two configurations and of their states of variables. Those assumptions are needed in order to show that there exist successors of both configurations by symbolic execution of the sequence of labels.

lemma *subsums-imp-feasible* :
  assumes *finite-labels ls*
  assumes *finite* (*pred* c_1)
  assumes *finite* (*pred* c_2)
  assumes $\forall\ e \in$ *pred* c_1. *finite* (*Bexp.vars e*)
  assumes $\forall\ e \in$ *pred* c_2. *finite* (*Bexp.vars e*)
  assumes $c_2 \sqsubseteq c_1$
  assumes *feasible* c_2 *ls*
  shows *feasible* c_1 *ls*
using *assms*
proof (*induct ls arbitrary* : c_1 c_2)
  case *Nil* **thus** *?case* **by** (*simp add* : *feasible-def sat-sub-by-sat*)
next
  case (*Cons l ls* c_1 c_2)

  then obtain $c_2{}'$ **where** *SE* c_2 *l* $c_2{}'$
        and *sat* $c_2{}'$
        and *feasible* $c_2{}'$ *ls*
  using *feasible-Cons* **by** *blast*

  obtain $c_1{}'$ **where** *SE* c_1 *l* $c_1{}'$
  using *finite-conj*[*OF Cons*(*3*,*5*)]
     finite-pred-imp-SE-updatable
     updatable-imp-ex-SE-succ
  by *blast*

  moreover
  hence *sat* $c_1{}'$
  using *SE-mono-for-sub*[*OF* - ‹*SE* c_2 *l* $c_2{}'$› *Cons*(*7*)]
     sat-sub-by-sat[*OF* ‹*sat* $c_2{}'$›]
  by *fast*

  moreover
  have *feasible* $c_1{}'$ *ls*
  proof −

    have *finite-label l*
    and *finite-labels ls* **using** *Cons*(*2*) **by** *simp-all*

    have *finite* (*pred* $c_1{}'$)
    by (*rule SE-preserves-finiteness2*[*OF* ‹*SE* c_1 *l* $c_1{}'$› *Cons*(*3*)])

    moreover
    have *finite* (*pred* $c_2{}'$)
    by (*rule SE-preserves-finiteness2*[*OF* ‹*SE* c_2 *l* $c_2{}'$› *Cons*(*4*)])

moreover
have $\forall\, e{\in}pred\ c_1{}'.\ finite\ (Bexp.vars\ e)$
by (*rule SE-preserves-finiteness1*[*OF* ‹*finite-label l*› ‹*SE* c_1 *l* $c_1{}'$› *Cons*(*5*)])

moreover
have $\forall\, e{\in}pred\ c_2{}'.\ finite\ (Bexp.vars\ e)$
by (*rule SE-preserves-finiteness1*[*OF* ‹*finite-label l*› ‹*SE* c_2 *l* $c_2{}'$› *Cons*(*6*)])

moreover
have $c_2{}' \sqsubseteq c_1{}'$
by (*rule SE-mono-for-sub*[*OF* ‹*SE* c_1 *l* $c_1{}'$› ‹*SE* c_2 *l* $c_2{}'$› *Cons*(*7*)])

ultimately
show *?thesis* **using** *Cons*(*1*) ‹*feasible* $c_2{}'$ *ls*› ‹*finite-labels ls*› **by** *fast*
qed

ultimately
show *?case* **by** (*auto simp add* : *feasible-Cons*)
qed

A.6.9 Concrete execution

We illustrate our notion of symbolic execution by relating it with *CE*, an inductive predicate describing concrete execution. Unlike symbolic execution, concrete execution describes program behavior given program states, i.e. concrete valuations for program variables. The goal of this section is to show that our notion of symbolic execution is correct, that is: given two configurations such that one results from the symbolic execution of a sequence of labels from the other, then the resulting configuration represents the set of states that are reachable by concrete execution from the states of the original configuration.

inductive *CE* ::
('*v*,'*d*) *state* ⇒ ('*v*,'*d*) *label* ⇒ ('*v*,'*d*) *state* ⇒ *bool*
where
CE σ *Skip* σ
| $e\ \sigma \Longrightarrow$ *CE* σ (*Assume e*) σ
| *CE* σ (*Assign v e*) ($\sigma(v := e\ \sigma)$)

inductive *CE-star* :: ('*v*,'*d*) *state* ⇒ ('*v*,'*d*) *label list* ⇒ ('*v*,'*d*) *state* ⇒ *bool* **where**
CE-star c [] *c*
| *CE* c_1 *l* $c_2 \Longrightarrow$ *CE-star* c_2 *ls* $c_3 \Longrightarrow$ *CE-star* c_1 (*l* # *ls*) c_3

lemma [*simp*] :
CE σ *Skip* $\sigma' = (\sigma' = \sigma)$
by (*auto simp add* : *CE.simps*)

```
lemma [simp] :
  CE σ (Assume e) σ' = (σ' = σ ∧ e σ)
by (auto simp add : CE.simps)


lemma [simp] :
  CE σ (Assign v e) σ' = (σ' = σ(v := e σ))
by (auto simp add : CE.simps)


lemma SE-as-CE :
  assumes SE c l c'
  shows   states c' = {σ'. ∃ σ ∈ states c. CE σ l σ'}
using assms
by (cases l)
   (auto simp add: states-of-SE-assume states-of-SE-assign)


lemma [simp] :
  CE-star σ [] σ' = (σ' = σ)
by (subst CE-star.simps) simp


lemma CE-star-Cons :
  CE-star σ1 (l # ls) σ2 = (∃ σ. CE σ1 l σ ∧ CE-star σ ls σ2)
by (subst (1) CE-star.simps) blast


lemma SE-star-as-CE-star :
  assumes SE-star c ls c'
  shows   states c' = {σ'. ∃ σ ∈ states c. CE-star σ ls σ'}
using assms
proof (induct ls arbitrary : c)
  case Nil thus ?case by simp
next
  case (Cons l ls c)

  then obtain c'' where SE c l c''
                 and   SE-star c'' ls c'
  using SE-star-Cons by blast

  show ?case
  unfolding set-eq-iff Bex-def mem-Collect-eq
  proof (intro allI iffI, goal-cases)
    case (1 σ')

    then obtain σ'' where σ'' ∈ states c''
```

```
          and   CE-star σ'' ls σ'
  using Cons(1) ‹SE-star c'' ls c'› by blast

  moreover
  then obtain σ where σ ∈ states c
         and   CE σ l σ''
  using ‹SE c l c''›  SE-as-CE by blast

  ultimately
  show ?case by (simp add: CE-star-Cons) blast
 next
  case (2 σ')

  then obtain σ where σ ∈ states c
         and   CE-star σ (l#ls) σ'
  by blast

  moreover
  then obtain σ'' where CE σ l σ''
          and   CE-star σ'' ls σ'
  using CE-star-Cons by blast

  ultimately
  show ?case
  using Cons(1) ‹SE-star c'' ls c'› ‹SE c l c''› by (auto simp add : SE-as-CE)
 qed
qed
```

A.6.10 Weakest Precondition Calculus

We model weakest precondition calculus by the following inductive predicate. In practice, we use *WP* to compute safeguard conditions.

```
fun WP ::
 ('v,'d) label ⇒ ('v,'d) bexp ⇒ ('v,'d) bexp
where
  WP Skip e = e

| WP (Assume e') e = (λ σ. ¬ e' σ ∨ e σ)

| WP (Assign v e') e = (λ σ. e (σ(v := e' σ)))


lemma
 shows WP (Assign x (λ σ::('v ⇒ nat). σ x + 1))
          (λ σ. σ x ≥ 0)
        = (λ σ. σ x + 1 ≥ 0)
by auto
```

The following property of *WP* ensures that we label red vertices (see

RB.thy) with safeguard conditions that are entailed by their configuration.

```
lemma
  assumes SE c l c′
  assumes c′ ⊨c φ
  shows   c ⊨c WP l φ
using assms
by (cases l) (auto simp add : sem-def entails-def states-of-SE-assume states-of-SE-assign)

end
theory Graph
imports Main
begin
```

A.7 Rooted Graphs

In this section, we model rooted graphs and their sub-paths and paths. We give a number of lemmas that will help proofs in the following theories, but that are very specific to our approach.

First, we will need the following simple lemma, which is not graph related, but that will prove useful when we will want to exhibit the last element of a non-empty sequence.

```
lemma neq-Nil-conv2 :
  xs ≠ [] = (∃ x xs′. xs = xs′ @ [x])
by (induct xs rule : rev-induct, auto)
```

A.7.1 Basic definitions and properties

Edges

We model edges by a record *′v edge* which is parameterized by the type *′v* of vertices. This allows us to represent the red part of red-black graphs as well as the black part (i.e. LTS) using extensible records (more on this later). Edges have two components, *src* and *tgt*, which respectively give their source and target.

```
record ′v edge =
  src  :: ′v
  tgt  :: ′v
```

Rooted graphs

We model rooted graphs by the record *′v rgraph*. It consists of two components: its root and its set of edges.

```
record ′v rgraph =
  root  :: ′v
  edges :: ′v edge set
```

Vertices

The set of vertices of a rooted graph is made of its root and the endpoints of its edges. Adding a vertex set component to the $'v$ *rgraph* record would require to assume, in the following, that the ends of edges of a rooted graph are elements of this set. Instead, we choose to deduce the set of vertices of a graph from its two components. The previous property is implicitly assumed.

Isabelle/HOL provides *extensible records*, i.e. it is possible to define records using existing ones by adding components. The following definition suppose that g is of type $('v,'x)$ *rgraph-scheme*, i.e. an object that has at least all the components of a $'v$ *rgraph*. The second type parameter $'x$ stands for the hypothetical type parameters that such an object could have in addition of the type of vertices $'v$. Using $('v,'x)$ *rgraph-scheme* instead of $'v$ *rgraph* allows to reuse the following definition(s) for all type of objects that have at least the components of a rooted graph. For example, we will reuse the following definition to characterize the set of locations of a LTS (see `LTS.thy`).

definition *vertices* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *set*
where
  vertices g $=$ $\{$*root g*$\}$ $\cup$ *src* ‘*edges g* $\cup$ *tgt* ‘ *edges g*

Basic properties of rooted graphs

In the following, we will be only interested in loop free rooted graphs and in what we call *well formed rooted graphs*. A well formed rooted graph is rooted graph that either has an empty set of edges or has at least one edge whose source is its root.

abbreviation *wf-rgraph* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
wf-rgraph g $\equiv$ *root g* $\in$ *src* ‘ *edges g* $=$ (*edges g* $\neq$ $\{\}$)

Although we are only interested in this kind of rooted graphs, we will only assume a graph is well formed when needed.

Out-going edges

This abbreviation will prove handy in the following.

abbreviation *out-edges* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ *edge set*
where
  out-edges g v $\equiv$ $\{$*e* $\in$ *edges g*. *src e* $=$ *v*$\}$

A.7.2 Consistent edge sequences, sub-paths and paths

Consistency of a sequence of edges

A sequence of edges *es* is consistent from vertex v_1 to another vertex v_2 if $v_1 = v_2$ if it either is empty or if:

- v_1 is the source of its first element, and
- v_2 is the target of its last element, and
- the target of each of its elements is the source of the following one.

```
fun ces ::
  'v ⇒ 'v edge list ⇒ 'v ⇒ bool
where
  ces v₁ [] v₂ = (v₁ = v₂)
| ces v₁ (e#es) v₂ = (src e = v₁ ∧ ces (tgt e) es v₂)
```

Sub-paths and paths

Let g be a rooted graph, *es* a sequence of edges and v_1 and v_2 two vertices. *es* is a sub-path in g from v_1 to v_2 if:

- it is consistent from v_1 to v_2,
- v_1 is a vertex of g,
- all of its elements are edges of g.

The second constraint is needed in the case of the empty sequence: without it, the empty sequence would be a sub-path of g even when *v1* is not one of its vertices. We do not require v_2 to be a vertex of g: this is a direct consequence of the definition.

```
definition subpath ::
  ('v,'x) rgraph-scheme ⇒ 'v ⇒ 'v edge list ⇒ 'v ⇒ bool
where
  subpath g v₁ es v₂ ≡ ces v₁ es v₂ ∧ v₁ ∈ vertices g ∧ set es ⊆ edges g
```

Although trivial, this lemma will help the simplifier in some future cases.

```
lemma fst-of-sp-is-vert :
  assumes subpath g v₁ es v₂
  shows   v₁ ∈ vertices g
using assms by (simp add : subpath-def)
```

As mentioned, the fact that the last vertex of a sub-path is a vertex of the considered graph is a direct consequence of the definition of *subpath*.

```
lemma lst-of-sp-is-vert :
  assumes subpath g v₁ es v₂
  shows   v₂ ∈ vertices g
```

using *assms* **by** (*induction es arbitrary* : v_1, *auto simp add*: *subpath-def vertices-def*)

In the following, we will not always be interested in the final vertex of a sub-path. We will use the abbreviation *subpath-from* whenever this final vertex has no importance, and *subpath* otherwise. We will also sometimes be interested in the set *subpaths-from* of sub-paths starting at a given vertex, which is defined by comprehension from *subpath-from*.

abbreviation *subpath-from* ::
 $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ *edge list* $\Rightarrow$ *bool*
where
 subpath-from g v es $\equiv \exists\ v'$. *subpath g v es* v'

abbreviation *subpaths-from* ::
 $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ *edge list set*
where
 subpaths-from g v $\equiv$ {*es*. *subpath-from g v es*}

A path is a sub-path starting at the root of the graph.

abbreviation *path* ::
 $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *edge list* $\Rightarrow$ $'v$ $\Rightarrow$ *bool*
where
 path g es v $\equiv$ *subpath g* (*root g*) *es v*

abbreviation *paths* ::
 $('a,'b)$ *rgraph-scheme* $\Rightarrow$ $'a$ *edge list set*
where
 paths g $\equiv$ {*es*. $\exists$ *v*. *path g es v*}

Some useful simplification lemmas for *subpath*.

lemma *sp-one* :
 subpath g v_1 [*e*] v_2 = (*src e* = v_1 $\wedge$ *e* $\in$ *edges g* $\wedge$ *tgt e* = v_2)
by (*auto simp add* : *subpath-def vertices-def*)

lemma *sp-Cons* :
 subpath g v_1 (*e*#*es*) v_2 = (*src e* = v_1 $\wedge$ *e* $\in$ *edges g* $\wedge$ *subpath g* (*tgt e*) *es* v_2)
by (*auto simp add* : *subpath-def vertices-def*)

lemma *sp-append-one* :
 subpath g v_1 (*es*@[*e*]) v_2 = (*subpath g* v_1 *es* (*src e*) $\wedge$ *e* $\in$ *edges g* $\wedge$ *tgt e* = v_2)
by (*induct es arbitrary* : v_1, *auto simp add* : *subpath-def vertices-def*)

lemma *sp-append* :
 subpath g v_1 (*es1*@*es2*) v_2 = ($\exists$ *v*. *subpath g* v_1 *es1 v* $\wedge$ *subpath g v es2* v_2)
by (*induct es1 arbitrary* : v_1)

((*simp add* : *subpath-def*, *fast*),
(*auto simp add* : *fst-of-sp-is-vert sp-Cons*))

A sub-path leads to a unique vertex.

lemma *sp-same-src-imp-same-tgt* :
assumes *subpath g v es* v_1
assumes *subpath g v es* v_2
shows $v_1 = v_2$
using *assms*
by (*induct es arbitrary* : *v*)
(*auto simp add* : *sp-Cons subpath-def vertices-def*)

In the following, we are interested in the evolution of the set of sub-paths of our symbolic execution graph after symbolic execution of a transition from the LTS representation of the program under analysis. Symbolic execution of a transition results in adding to the graph a new edge whose source is already a vertex of this graph, but not its target. The following lemma describes sub-paths ending in the target of such an edge.

Let *e* be an edge whose target has not out-going edges. A sub-path *es* containing *e* ends by *e* and this occurrence of *e* is unique along *es*.

lemma *sp-through-de-decomp* :
assumes *out-edges g* (*tgt e*) = {}
assumes *subpath g* v_1 *es* v_2
assumes $e \in$ *set es*
shows $\exists$ *es'*. *es* = *es'* @ [*e*] $\wedge$ $e \notin$ *set es'*
using *assms*(*2*,*3*)
proof (*induction es arbitrary* : v_1)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons e' es*)

hence $e = e' \vee (e \neq e' \wedge e \in$ *set es*) **by** *auto*

thus *?case*
proof (*elim disjE*, *goal-cases*)
case *1*

moreover
hence *es* = [] **using** *assms*(*1*) *Cons* **by** (*cases es*, *auto simp add*: *sp-Cons*)

ultimately
show *?case* **by** *auto*
next
case *2* **thus** *?case*
using *assms*(*1*) *Cons*(*1*)[*of tgt e'*] *Cons*(*2*)
by (*auto simp add* : *sp-Cons*)
qed
qed

A.7.3 Adding edges

This definition and the following lemma are here mainly to ease the definitions and proofs in the next theories.

```
abbreviation add-edge ::
  ('v,'x) rgraph-scheme ⇒ 'v edge ⇒ ('v,'x) rgraph-scheme
where
  add-edge g e ≡ rgraph.edges-update (λ edges. edges ∪ {e}) g
```

Let *es* be a sub-path from a vertex other than the target of *e* in the graph obtained from *g* by the addition of edge *e*. Moreover, assume that the target of *e* is not a vertex of *g*. Then *e* is an element of *es*.

```
lemma sp-ends-in-tgt-imp-mem :
  assumes tgt e ∉ vertices g
  assumes v ≠ tgt e
  assumes subpath (add-edge g e) v es (tgt e)
  shows   e ∈ set es
proof −
  have es ≠ [] using assms(2,3) by (auto simp add : subpath-def)

  then obtain e' es' where es = es' @ [e'] using neq-Nil-conv2[of es] by blast

  thus ?thesis using assms(1,3) by (auto simp add : sp-append-one vertices-def
image-def)
qed
```

```
end
theory LTS
imports Graph SymExec
begin
```

A.8 Labeled Transition Systems

This theory is motivated by the need of an abstract representation of control-flow graphs (CFG). It is a refinement of the prior theory of (unlabeled) graphs and proceeds by decorating their edges with *labels* expressing assumptions and effects (assignments) on an underlying state. In this theory, we define LTSs and introduce a number of abbreviations that will ease stating and proving lemmas in the following theories.

A.8.1 Basic definitions

The labeled transition systems (LTS) we are heading for are constructed by extending *rgraph*'s by a labeling function of the edges, using Isabelle

extensible records.

record $('vert,'var,'d)$ *lts* = *'vert rgraph* +
  labeling :: *'vert edge* ⇒ *('var,'d) label*

We call *initial location* the root of the underlying graph.

abbreviation *init* ::
('vert,'var,'d,'x) lts-scheme ⇒ *'vert*
where
  init lts ≡ *root lts*

The set of labels of a LTS is the image set of its labeling function over its set of edges.

abbreviation *labels* ::
  ('vert,'var,'d,'x) lts-scheme ⇒ *('var,'d) label set*
where
  labels lts ≡ *labeling lts* ‘ *edges lts*

In the following, we will sometimes need to use the notion of *trace* of a given sequence of edges with respect to the transition relation of an LTS.

abbreviation *trace* ::
  'vert edge list ⇒ *('vert edge* ⇒ *('var,'d) label)* ⇒ *('var,'d) label list*
where
  trace es L ≡ *map L es*

We are interested in a special form of Labeled Transition Systems; the prior record definition is too liberal. We will constrain it to *well-formed labeled transition systems.*

We first define an application that, given an LTS, returns its underlying graph.

abbreviation *graph* ::
  ('vert,'var,'d,'x) lts-scheme ⇒ *'vert rgraph*
where
  graph lts ≡ *rgraph.truncate lts*

An LTS is well-formed if its underlying *rgraph* is well-formed.

abbreviation *wf-lts* ::
  ('vert,'var,'d,'x) lts-scheme ⇒ *bool*
where
  wf-lts lts ≡ *(wf-rgraph lts)*

In the following theories, we will sometimes need to account for the fact that we consider LTSs with a finite number of edges.

abbreviation *finite-lts* ::
  ('vert,'var,'d,'x) lts-scheme ⇒ *bool*
where
  finite-lts lts ≡ ∀ *l* ∈ *range (labeling lts). finite-label l*

A.8.2 Feasible sub-paths and paths

A sequence of edges is a feasible sub-path of an LTS *lts* from a configuration *c* if it is a sub-path of the underlying graph of *lts* and if it is feasible from the configuration *c*.

abbreviation *feasible-subpath* ::
 $('vert,'var,'d,'x)$ *lts-scheme* ⇒
 $('var,'d)$ *conf* ⇒
 'vert ⇒
 'vert edge list ⇒
 'vert ⇒ *bool*
where
 feasible-subpath lts c v_1 *es* v_2 ≡ *Graph.subpath lts* v_1 *es* v_2
 ∧ *feasible c* (*trace es* (*labeling lts*))

Similarly to sub-paths in rooted raphs, we will not be always interested in the final vertex of a feasible sub-path. We use the following notion when we are not interested in this vertex.

abbreviation *feasible-subpath-from* ::
 $('vert,'var,'d,'x)$ *lts-scheme* ⇒ $('var,'d)$ *conf* ⇒ *'vert* ⇒ *'vert edge list* ⇒ *bool*
where
 feasible-subpath-from lts c v es ≡ ∃ *v'*. *feasible-subpath lts c v es v'*

abbreviation *feasible-subpaths-from* ::
 $('vert,'var,'d,'x)$ *lts-scheme* ⇒ $('var,'d)$ *conf* ⇒ *'vert* ⇒ *'vert edge list set*
where
 feasible-subpaths-from lts c v ≡ {*es*. *feasible-subpath-from lts c v es*}

As earlier, feasible paths are defined as feasible sub-paths starting at the initial location of the LTS.

abbreviation *feasible-path* ::
 $('vert,'var,'d,'x)$ *lts-scheme* ⇒ $('var,'d)$ *conf* ⇒ *'vert edge list* ⇒ *'vert* ⇒ *bool*
where
 feasible-path lts c es v ≡ *feasible-subpath lts c* (*init lts*) *es v*

abbreviation *feasible-paths* ::
 $('vert,'var,'d,'x)$ *lts-scheme* ⇒ $('var,'d)$ *conf* ⇒ *'vert edge list set*
where
 feasible-paths lts c ≡ {*es*. ∃ *v*. *feasible-path lts c es v*}

end
theory *SubRel*
imports *Graph*
begin

A.9 Graphs Equipped with Ssubsumption Relations

In this section, we define subsumption relations and the notion of sub-paths in rooted graphs equipped with such relations. We proceed in the same manner than in `Graph.thy`: first we define the consistency of a sequence of edges in presence of a subsumption relation, then sub-paths. We are interested in subsumptions taking places between red vertices of red-black graphs (see `RB.thy`), i.e. occurrences of locations of LTS. Here subsumptions are defined as pairs of indexed vertices of a LTS, and subsumption relations as sets of subsumptions. The type of vertices of such LTS is represented by the abstract type $'v$ in the following.

A.9.1 Basic definitions and properties

Subsumptions and subsumption relations

Subsumptions take place between occurrences of the vertices of a graph. We represent such occurrences by indexed versions of vertices. A subsumption is defined as pair of indexed vertices.

type-synonym $'v$ *sub-t* $= (('v \times nat) \times ('v \times nat))$

A subsumption relation is a set of subsumptions.

type-synonym $'v$ *sub-rel-t* $=$ $'v$ *sub-t set*

We consider the left member to be subsumed by the right one. The left member of a subsumption is called its *subsumee*, the right member its *subsumer*.

abbreviation *subsumee* ::
$'v$ *sub-t* $\Rightarrow ('v \times nat)$
where
subsumee sub $\equiv$ *fst sub*

abbreviation *subsumer* ::
$'v$ *sub-t* $\Rightarrow ('v \times nat)$
where
subsumer sub $\equiv$ *snd sub*

We will need to talk about the sets of subsumees and subsumers of a subsumption relation.

abbreviation *subsumees* ::
$'v$ *sub-rel-t* $\Rightarrow ('v \times nat)$ *set*
where
subsumees subs $\equiv$ *subsumee* ` *subs*

abbreviation *subsumers* ::
$'v$ *sub-rel-t* $\Rightarrow$ ($'v \times$ *nat*) *set*
where
subsumers subs $\equiv$ *subsumer* ‘ *subs*

The two following lemmas will prove useful in the following.

lemma *subsumees-conv* :
subsumees subs $= \{v.\ \exists\ v'.\ (v,v') \in$ *subs*$\}$
by *force*

lemma *subsumers-conv* :
subsumers subs $= \{v'.\ \exists\ v.\ (v,v') \in$ *subs*$\}$
by *force*

We call set of vertices of the relation the union of its sets of subsumees and subsumers.

abbreviation *vertices* ::
$'v$ *sub-rel-t* $\Rightarrow$ ($'v \times$ *nat*) *set*
where
vertices subs $\equiv$ *subsumers subs* $\cup$ *subsumees subs*

A.9.2 Well-formed subsumption relation of a graph

Well-formed subsumption relations

In the following, we make an intensive use of *locales*. We use them as a convenient way to add assumptions to the following lemmas, in order to ease their reading. Locales can be built from locales, allowing some modularity in the formalization. The following locale simply states that we suppose there exists a subsumption relation called *subs*. It will then be used to constrain subsumption relations.

locale *sub-rel* =
fixes *subs* :: $'v$ *sub-rel-t*

We are only interested in subsumptions involving two different occurrences of the same LTS location. Moreover, once a vertex has been subsumed, there is no point in trying to subsume it again by another subsumer: subsumees must have a unique subsumer. Finally, we do not allow chains of subsumptions, thus the intersection of the sets of subsumers and subsumees must be empty. Such subsumption relations are said to be *well-formed*.

locale *wf-sub-rel* = *sub-rel* +
assumes *sub-imp-same-verts* :
sub $\in$ *subs* $\Longrightarrow$ *fst* (*subsumee sub*) = *fst* (*subsumer sub*)

assumes *subsumed-by-one* :
$\forall\ v \in$ *subsumees subs*. $\exists!\ v'.\ (v,v') \in$ *subs*

```
  assumes inter-empty :
    subsumers subs ∩ subsumees subs = {}

begin
  lemmas wf-sub-rel = sub-imp-same-verts subsumed-by-one inter-empty
```

A rephrasing of the assumption *subsumed-by-one*.

```
  lemma (in wf-sub-rel) subsumed-by-two-imp :
    assumes (v,v₁) ∈ subs
    assumes (v,v₂) ∈ subs
    shows   v₁ = v₂
  using assms wf-sub-rel unfolding subsumees-conv by blast
```

A well-formed subsumption is its own transitive closure.

```
  lemma in-trancl-imp :
    assumes (v,v') ∈ subs⁺
    shows   (v,v') ∈ subs
  using tranclD[OF assms] tranclD[of - v' subs]
        rtranclD[of - v' subs]
        inter-empty
  by force

  lemma trancl-eq :
    subs⁺ = subs
  using in-trancl-imp r-into-trancl by fast
end
```

Subsumption relation of a graph

We consider subsumption relations to equip rooted graphs. However, nothing in the previous definitions relates these relations to graphs: subsumptions relations involve objects that are of the type of indexed vertices, but that might to not be vertices of an actual graph. We equip graphs with subsumption relations using the notion of *sub-relation of a graph*. Such a relation must only involves vertices of the graph it equips.

```
locale rgraph =
  fixes g :: ('v,'x) rgraph-scheme


locale sub-rel-of = rgraph + sub-rel +
  assumes related-are-verts : vertices subs ⊆ Graph.vertices g
begin
  lemmas sub-rel-of = related-are-verts
```

The transitive closure of a sub-relation of a graph g is also a sub-relation of g.

```
  lemma trancl-sub-rel-of :
```

```
  sub-rel-of g (subs⁺)
 using tranclD[of - - subs] tranclD2[of - - subs] sub-rel-of
 unfolding sub-rel-of-def subsumers-conv subsumees-conv by blast
end
```

Well-formed sub-relations

We pack both previous locales into a third one. We talk about *well-formed sub-relations*.

```
locale wf-sub-rel-of = rgraph + sub-rel +
 assumes sub-rel-of : sub-rel-of g subs
 assumes wf-sub-rel : wf-sub-rel subs
begin
 lemmas wf-sub-rel-of = sub-rel-of wf-sub-rel
end
```

As previously, even if we are only interested by well-formed sub-relations, we assume the relation is such only when needed.

A.9.3 Consistent edge sequences and sub-paths

Consistency in presence of a subsumption relation

We model sub-paths in the same spirit than in `Graph.thy`, by starting with defining the consistency of a sequence of edges w.r.t. a subsumption relation. The idea is that subsumption links can "fill the gaps" between subsequent edges that would have made the sequence inconsistent otherwise. For now, we define consistency of a sequence w.r.t. any subsumption relation. Thus, we cannot account yet for the fact that we only consider relations without chains of subsumptions. The empty sequence is consistent w.r.t. to a subsumption relation from v_1 to v_2 if these two vertices are equal or if they belong to the transitive closure of the relation. A non-empty sequence is consistent if it is made of consistent sequences whose extremities are linked in the transitive closure of the subsumption relation.

```
fun ces ::
  ('v × nat) ⇒ ('v × nat) edge list ⇒ ('v × nat) ⇒ 'v sub-rel-t ⇒ bool
where
  ces v₁ [] v₂ subs = (v₁ = v₂ ∨ (v₁,v₂) ∈ subs⁺)
| ces v₁ (e#es) v₂ subs = ((v₁ = src e ∨ (v₁,src e) ∈ subs⁺) ∧ ces (tgt e) es v₂
subs)
```

A consistent sequence from v_1 to v_2 without a subsumption relation is consistent between these two vertices in presence of any relation.

```
lemma
 assumes Graph.ces v₁ es v₂
 shows   ces v₁ es v₂ subs
using assms by (induct es arbitrary : v₁, auto)
```

Consistency in presence of the empty subsumption relation reduces to consistency as defined in `Graph.thy`.

lemma
  assumes *ces* v_1 *es* v_2 {}
  shows *Graph.ces* v_1 *es* v_2
using *assms* **by** (*induct es arbitrary* : v_1, *auto*)

Let (v_1, v_2) be an element of a subsumption relation, and *es* a sequence of edges consistent w.r.t. this relation from vertex v_2. Then *es* is also consistent from v_1. Even if this lemma will not be used in the following, this is the base fact for saying that paths feasible from a subsumee are also feasible from its subsumer.

lemma
  assumes (v_1,v_2) ∈ *subs*
  assumes *ces* v_2 *es v subs*
  shows *ces* v_1 *es v subs*
using *assms* **by** (*cases es*, *simp-all*) (*intro disjI2*, *force*)+

Let *es* be a sequence of edges consistent w.r.t. a subsumption relation. Extending this relation preserves the consistency of *es*.

lemma *ces-Un* :
  assumes *ces* v_1 *es* v_2 $subs_1$
  shows *ces* v_1 *es* v_2 ($subs_1$ ∪ $subs_2$)
using *assms* **by** (*induct es arbitrary* : v_1, *auto simp add* : *trancl-mono*)

Simplification lemmas for *SubRel.ces*.

lemma *ces-append-one* :
  ces v_1 (*es* @ [*e*]) v_2 *subs* = (*ces* v_1 *es* (*src e*) *subs* ∧ *ces* (*src e*) [*e*] v_2 *subs*)
by (*induct es arbitrary* : v_1, *auto*)

lemma *ces-append* :
  ces v_1 (es_1 @ es_2) v_2 *subs* = (∃ *v*. *ces* v_1 es_1 *v subs* ∧ *ces v* es_2 v_2 *subs*)
proof (*intro iffI*, *goal-cases*)
  case *1* **thus** *?case*
  by (*induct* es_1 *arbitrary* : v_1)
    (*simp-all del* : *split-paired-Ex*, *blast*)
next
  case *2* **thus** *?case*
  proof (*induct* es_1 *arbitrary* : v_1)
    case (*Nil* v_1)

    then obtain *v* **where** *ces* v_1 [] *v subs*
        and *ces v* es_2 v_2 *subs*
    by *blast*

    thus *?case*
    unfolding *ces.simps*

```
  proof (elim disjE, goal-cases)
    case 1 thus ?case by simp
  next
    case 2 thus ?case by (cases es₂) (simp, intro disjI2, fastforce)+
  qed
next
  case Cons thus ?case by auto
qed
qed
```

Let *es* be a sequence of edges consistent from v_1 to v_2 w.r.t. a sub-relation *subs* of a graph *g*. Suppose elements of this sequence are edges of *g*. If v_1 is a vertex of *g* then v_2 is also a vertex of *g*.

```
lemma (in sub-rel-of) ces-imp-end-vertex :
  assumes ces v₁ es v₂ subs
  assumes set es ⊆ edges g
  assumes v₁ ∈ Graph.vertices g
  shows   v₂ ∈ Graph.vertices g
using assms trancl-sub-rel-of
unfolding sub-rel-of-def subsumers-conv vertices-def
by (induct es arbitrary : v₁) (force, (simp del : split-paired-Ex, fast))
```

Sub-paths

A sub-path leading from v_1 to v_2, two vertices of a graph *g* equipped with a subsumption relation *subs*, is a sequence of edges consistent w.r.t. *subs* from v_1 to v_2 whose elements are edges of *g*. Moreover, we must assume that *subs* is a sub-relation of *g*, otherwise *es* could “exit” *g* through subsumption links.

Once again, the fact that v_2 is a vertex of *g* is implied by the following definition.

```
definition subpath ::
  (('v × nat),'x) rgraph-scheme ⇒
   ('v × nat)                  ⇒
   ('v × nat) edge list         ⇒
   ('v × nat)                  ⇒
   (('v × nat) × ('v × nat)) set ⇒ bool
where
  subpath g v₁ es v₂ subs ≡ sub-rel-of g subs
                          ∧ v₁ ∈ Graph.vertices g
                          ∧ ces v₁ es v₂ subs
                          ∧ set es ⊆ edges g
```

```
abbreviation path ::
  (('v × nat),'x) rgraph-scheme ⇒
   ('v × nat) edge list         ⇒
   ('v × nat)                  ⇒
```

$(('v \times nat) \times ('v \times nat))$ *set* $\Rightarrow$ *bool*
where
path g es v subs $\equiv$ *subpath g* (*root g*) *es v subs*

This lemma will ease the proofs of some goals in the following.

lemma *fst-of-sp-is-vert* :
assumes *subpath g* v_1 *es* v_2 *subs*
shows $v_1 \in$ *Graph.vertices g*
using *assms* **by** (*simp add* : *subpath-def*)

lemma *lst-of-sp-is-vert* :
assumes *subpath g* v_1 *es* v_2 *subs*
shows $v_2 \in$ *Graph.vertices g*
using *assms sub-rel-of.ces-imp-end-vertex* **unfolding** *subpath-def* **by** *fast*

Once again, in some cases, we will not be interested in the ending vertex of a sub-path.

abbreviation *subpath-from* ::
$(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow$ $('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow$ *'v sub-rel-t* $\Rightarrow$ *bool*
where
subpath-from g v es subs $\equiv \exists$ v'. *subpath g v es* v' *subs*

Simplification lemmas for *SubRel.subpath*.

lemma *Nil-sp* :
subpath g v_1 [] v_2 *subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ $v_1 \in$ *Graph.vertices g*
$\wedge$ $(v_1 = v_2 \vee (v_1,v_2) \in subs^+)$
by (*auto simp add* : *subpath-def*)

When the subsumption relation is well-formed (denoted by (*in wf-sub-rel*)), there is no need to account for the transitive closure of the relation. We will need both versions in the following.

lemma (**in** *wf-sub-rel*) *Nil-sp* :
subpath g v_1 [] v_2 *subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ $v_1 \in$ *Graph.vertices g*
$\wedge$ $(v_1 = v_2 \vee (v_1,v_2) \in subs)$
using *trancl-eq* **by** (*simp add* : *Nil-sp*)

Simplification lemma for the one-element sequence.

lemma *sp-one* :
shows *subpath g* v_1 [*e*] v_2 *subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ $(v_1 = src\ e \vee (v_1,src\ e) \in subs^+)$
$\wedge$ $e \in$ *edges g*
$\wedge$ $(tgt\ e = v_2 \vee (tgt\ e,v_2) \in subs^+)$
using *sub-rel-of.trancl-sub-rel-of*[*of g subs*]
by (*intro iffI*, *auto simp add* : *vertices-def sub-rel-of-def subpath-def*)

Once again, when the subsumption relation is well-formed, the previous lemma can be simplified since, in this case, the transitive closure of the relation is the relation itself.

lemma (**in** *wf-sub-rel*) *sp-one* :
shows *subpath* g v_1 [e] v_2 *subs* $\longleftrightarrow$ *sub-rel-of* g *subs*
$\wedge$ (v_1 = *src* e $\vee$ (v_1,*src* e) $\in$ *subs*)
$\wedge$ e $\in$ *edges* g
$\wedge$ (*tgt* e = v_2 $\vee$ (*tgt* e,v_2) $\in$ *subs*)
using *sp-one trancl-eq* **by** *fast*

Simplification lemma for the non-empty sequence (which might contain more than one element).

lemma *sp-Cons* :
shows *subpath* g v_1 (e # *es*) v_2 *subs* $\longleftrightarrow$ *sub-rel-of* g *subs*
$\wedge$ (v_1 = *src* e $\vee$ (v_1,*src* e) $\in$ *subs*$^+$)
$\wedge$ e $\in$ *edges* g
$\wedge$ *subpath* g (*tgt* e) *es* v_2 *subs*
using *sub-rel-of.trancl-sub-rel-of*[*of* g *subs*]
by (*intro iffI*, *auto simp add* : *subpath-def vertices-def sub-rel-of-def*)

The same lemma when the subsumption relation is well-formed.

lemma (**in** *wf-sub-rel*) *sp-Cons* :
subpath g v_1 (e # *es*) v_2 *subs* $\longleftrightarrow$ *sub-rel-of* g *subs*
$\wedge$ (v_1 = *src* e $\vee$ (v_1,*src* e) $\in$ *subs*)
$\wedge$ e $\in$ *edges* g
$\wedge$ *subpath* g (*tgt* e) *es* v_2 *subs*
using *sp-Cons trancl-eq* **by** *fast*

Simplification lemma for *SubRel.subpath* when the sequence is known to end by a given edge.

lemma *sp-append-one* :
subpath g v_1 (*es* @ [e]) v_2 *subs* $\longleftrightarrow$ *subpath* g v_1 *es* (*src* e) *subs*
$\wedge$ e $\in$ *edges* g
$\wedge$ (*tgt* e = v_2 $\vee$ (*tgt* e, v_2) $\in$ *subs*$^+$)
unfolding *subpath-def* **by** (*auto simp add* : *ces-append-one*)

Simpler version in the case of a well-formed subsumption relation.

lemma (**in** *wf-sub-rel*) *sp-append-one* :
subpath g v_1 (*es* @ [e]) v_2 *subs* $\longleftrightarrow$ *subpath* g v_1 *es* (*src* e) *subs*
$\wedge$ e $\in$ *edges* g
$\wedge$ (*tgt* e = v_2 $\vee$ (*tgt* e, v_2) $\in$ *subs*)
using *sp-append-one trancl-eq* **by** *fast*

Simplification lemma when the sequence is known to be the concatenation of two sub-sequences.

lemma *sp-append* :
subpath g v_1 (es_1 @ es_2) v_2 *subs* $\longleftrightarrow$
($\exists$ v. *subpath* g v_1 es_1 v *subs* $\wedge$ *subpath* g v es_2 v_2 *subs*)

proof (*intro iffI*, *goal-cases*)
  case *1* **thus** *?case*
  using *sub-rel-of.ces-imp-end-vertex*
  by (*simp add* : *subpath-def ces-append*) *blast*
next
  case *2* **thus** *?case*
  unfolding *subpath-def*
  by (*simp only* : *ces-append*) *fastforce*
qed

A sub-path ending in a subsumed vertex can be extended to the subsumer of this vertex.

lemma *sp-append-sub* :
  assumes *subpath g* v_1 *es* v_2 *subs*
  assumes $(v_2,v_3) \in$ *subs*
  shows *subpath g* v_1 *es* v_3 *subs*
proof (*cases es*)
  case *Nil*

  moreover
  hence $v_1 \in$ *Graph.vertices g*
  and $v_1 = v_2 \vee (v_1,v_2) \in subs^+$
  using *assms*(*1*) **by** (*simp-all add* : *Nil-sp*)

  ultimately
  show *?thesis*
  using *assms*(*1*,*2*)
    Nil-sp[*of g* v_1 v_2 *subs*]
    trancl-into-trancl[*of* v_1 v_2 *subs* v_3]
  by (*auto simp add* : *subpath-def*)
next
  case *Cons*

  then obtain *es′ e* **where** *es* = *es′* @ [*e*] **using** *neq-Nil-conv2*[*of es*] **by** *blast*

  thus *?thesis* **using** *assms trancl-into-trancl* **by** (*simp add* : *sp-append-one*) *fast*
qed

We consider a graph equipped with a subsumption relation containing an element $(v_1,\ v_2)$ where v_1 has not out-going edges and v_2 as unique subsumer, the latter not being subsumed itself. A sub-path starting in v_1 is either empty or can be considered to start in v_2.

lemma *sp-from-subsumee* :
  assumes $(v_1,v_2) \in$ *subs*
  assumes *subpath g* v_1 *es v subs*
  assumes *out-edges g* $v_1 = \{\}$
  assumes $\exists!\ v.\ (v_1,v) \in$ *subs*
  assumes $v_2 \notin$ *subsumees subs*
  shows *es* = [] $\vee$ *subpath g* v_2 *es v subs*

proof (*cases es*)
  case *Nil* **thus** *?thesis* **by** *simp*
next
  case (*Cons e es'*)

  show *?thesis*
  unfolding *Cons sp-Cons*
  proof (*rule disjI2*, *intro conjI*, *goal-cases*)
    case *1* **show** *?case* **using** *assms*(*2*) **by** (*simp add*: *subpath-def*)
  next
    case *2*

    have (v_2, *src e*) $\in$ *subs** **using** *assms*(*1*−*4*) *tranclD* **unfolding** *Cons sp-Cons*
by *fast*

    thus *?case* **using** *assms*(*3*) *rtranclD*[*of* v_2 *src e subs*] **by** *fast*
  next
    case *3* **show** *?case* **using** *assms*(*2*) **unfolding** *Cons sp-Cons* **by** *simp*
  next
    case *4* **show** *?case* **using** *assms*(*2*) **unfolding** *Cons sp-Cons* **by** *simp*
  qed
qed

Note that we could have used two lemmas instead of one, in order to split the two conclusions: the right disjunct might not be true if $v = v_1$.

We extend the previous lemma to well-formed subsumption relations. We will need both versions in the following.

lemma (**in** *wf-sub-rel*) *sp-from-subsumee* :
  assumes (v_1,v_2) $\in$ *subs*
  assumes *subpath g* v_1 *es v subs*
  assumes *out-edges g* v_1 = {}
  shows *es* = [] $\vee$ *subpath g* v_2 *es v subs*
using *assms*
proof −
  have $\exists$! *v*. (v_1,*v*) $\in$ *subs*
  and $v_2 \notin$ *subsumees subs*
  using *wf-sub-rel assms*(*1*) **unfolding** *subsumees-conv subsumers-conv* **by** *fast+*

  thus *?thesis* **by** (*rule sp-from-subsumee*[*OF assms*])
qed

A sub-path starting at a non-subsumed vertex whose set of out-edges is empty is also empty.

lemma *sp-from-de-empty* :
  assumes $v_1 \notin$ *subsumees subs*
  assumes *out-edges g* v_1 = {}
  assumes *subpath g* v_1 *es* v_2 *subs*
  shows *es* = []

```
using assms tranclD by (cases es) (auto simp add : sp-Cons, force)
```

Let e be an edge whose target is not subsumed and has no out-going edges. A sub-path es containing e ends by e and this occurrence of e is unique along es.

```
lemma sp-through-de-decomp :
  assumes tgt e ∉ subsumees subs
  assumes out-edges g (tgt e) = {}
  assumes subpath g v1 es v2 subs
  assumes e ∈ set es
  shows  ∃ es'. es = es' @ [e] ∧ e ∉ set es'
using assms(3,4)
proof (induction es arbitrary : v1)
  case (Nil v1) thus ?case by simp
next
  case (Cons e' es v1)

  hence subpath g (tgt e') es v2 subs
  and  e = e' ∨ (e ≠ e' ∧ e ∈ set es) by (auto simp add : sp-Cons)

  thus ?case
  proof (elim disjE, goal-cases)
    case 1 thus ?case
    using sp-from-de-empty[OF assms(1,2)] by fastforce
  next
    case 2 thus ?case using Cons(1)[of tgt e'] by force
  qed
qed
```

Consider a sub-path ending at the target of a recently added edge e, whose target did not belong to the graph prior to its addition. If es starts in another vertex than the target of e, then it contains e.

```
lemma (in sub-rel-of) sp-ends-in-tgt-imp-mem :
  assumes tgt e ∉ Graph.vertices g
  assumes v ≠ tgt e
  assumes subpath (add-edge g e) v es (tgt e) subs
  shows  e ∈ set es
proof −
  have tgt e ∉ subsumers subs using assms(1) sub-rel-of by auto

  hence (v,tgt e) ∉ subs+ using tranclD2 by force

  hence es ≠ [] using assms(2,3) by (auto simp add : Nil-sp)

  then obtain es' e' where es = es' @ [e'] using neq-Nil-conv2[of es] by blast

  moreover
  hence e' ∈ edges (add-edge g e) using assms(3) by (auto simp add: subpath-def)
```

moreover
have *tgt* $e' = tgt\ e$
using *tranclD2 assms*(*3*) ‹*tgt* $e \notin$ *subsumers subs*› ‹$es = es'$ @ [e']›
by (*force simp add* : *sp-append-one*)

ultimately
show *?thesis* **using** *assms*(*1*) **unfolding** *vertices-def image-def* **by** *force*
qed

end
theory *ArcExt*
imports *SubRel*
begin

A.10 Extending Graphs with Edges

In this section, we formalize the operation of adding to a rooted graph an edge whose source is already a vertex of the given graph but not its target. We call this operation an extension of the given graph by adding an edge. This corresponds to an abstraction of the act of adding an edge to the red part of a red-black graph as a result of symbolic execution of the corresponding transition in the LTS under analysis, where all details about symbolic execution would have been abstracted. We then state and prove a number of facts describing the evolution of the set of paths of the given graph, first without considering subsumption links then in the case of rooted graph equipped with a subsumption relation.

A.10.1 Definition and basic properties

Extending a rooted graph with an edge consists in adding to its set of edges an edge whose source is a vertex of this graph but whose target is not.

abbreviation *extends* ::
  ($'v$,$'x$) *rgraph-scheme* $\Rightarrow$ $'v$ *edge* $\Rightarrow$ ($'v$,$'x$) *rgraph-scheme* $\Rightarrow$ *bool*
where
  extends $g\ e\ g' \equiv src\ e \in$ *Graph.vertices* g
    $\wedge$ *tgt* $e \notin$ *Graph.vertices* g
    $\wedge$ $g' =$ (*add-edge* $g\ e$)

After such an extension, the set of out-edges of the target of the new edge is empty.

lemma *extends-tgt-out-edges* :
  assumes *extends* $g\ e\ g'$
  shows *out-edges* g' (*tgt* e) $= \{\}$
using *assms* **unfolding** *vertices-def image-def* **by** *force*

Consider a graph equipped with a sub-relation. This relation is also a sub-relation of any extension of this graph.

lemma (**in** *sub-rel-of*)
  assumes *extends g e g'*
  shows *sub-rel-of g' subs*
using *assms sub-rel-of* **by** (*auto simp add* : *sub-rel-of-def vertices-def*)

Extending a graph with an edge preserves the existing sub-paths.

lemma *sp-in-extends* :
  assumes *extends g e g'*
  assumes *Graph.subpath g* v_1 *es* v_2
  shows *Graph.subpath g'* v_1 *es* v_2
using *assms* **by** (*auto simp add* : *Graph.subpath-def vertices-def*)

A.10.2 Properties of sub-paths in an extension

Extending a graph by an edge preserves the existing sub-paths.

lemma *sp-in-extends-w-subs* :
  assumes *extends g a g'*
  assumes *subpath g* v_1 *es* v_2 *subs*
  shows *subpath g'* v_1 *es* v_2 *subs*
using *assms* **by** (*auto simp add* : *subpath-def sub-rel-of-def vertices-def*)

In an extension, the target of the new edge has no out-edges. Thus sub-paths of the extension starting and ending in old vertices are sub-paths of the graph prior to its extension.

lemma (**in** *sub-rel-of*) *sp-from-old-verts-imp-sp-in-old* :
  assumes *extends g e g'*
  assumes $v_1 \in$ *Graph.vertices g*
  assumes $v_2 \in$ *Graph.vertices g*
  assumes *subpath g'* v_1 *es* v_2 *subs*
  shows *subpath g* v_1 *es* v_2 *subs*
proof −
  have $e \notin$ *set es*
  proof (*intro notI*)
    assume $e \in$ *set es*

    have v_2 = *tgt e*
    proof −
      have *tgt e* $\notin$ *subsumees subs* **using** *sub-rel-of assms*(*1*) **by** *fast*

      moreover
     have *out-edges g'* (*tgt e*) = {} **using** *assms*(*1*) **by** (*rule extends-tgt-out-edges*)

      ultimately
      have ∃ *es'*. *es* = *es'* @ [*e*] ∧ *e* ∉ *set es'*
      using *assms*(*4*) ‹*e* ∈ *set es*› **by** (*intro sp-through-de-decomp*)

      then obtain *es'* **where** *es* = *es'* @ [*e*] *e* ∉ *set es'* **by** *blast*

      hence *tgt e* = v_2 ∨ (*tgt e*,v_2) ∈ $subs^+$

```
      using assms(4) by (simp add : sp-append-one)

     thus ?thesis using ‹tgt e ∉ subsumees subs› tranclD[of tgt e v2 subs] by force
    qed

    thus False using assms(1,3) by simp
  qed

  thus ?thesis
  using sub-rel-of assms
  unfolding subpath-def sub-rel-of-def by auto
qed
```

For the same reason, sub-paths starting at the target of the new edge are empty.

```
lemma (in sub-rel-of) sp-from-tgt-in-extends-is-Nil :
  assumes extends g e g'
  assumes subpath g' (tgt e) es v subs
  shows   es = []
using sub-rel-of assms
      extends-tgt-out-edges
      sp-from-de-empty[of tgt e subs g' es v]
by fast
```

Moreover, a sub-path es starting in another vertex than the target of the new edge e but ending in this target has e as last element. This occurrence of e is unique among es. The prefix of es preceding e is a sub-path leading at the source of e in the original graph.

```
lemma (in sub-rel-of) sp-to-new-edge-tgt-imp :
  assumes extends g e g'
  assumes subpath g' v es (tgt e) subs
  assumes v ≠ tgt e
  shows   ∃ es'. es = es' @ [e] ∧ e ∉ set es' ∧ subpath g v es' (src e) subs
proof −
  obtain es' where es = es' @ [e] and e ∉ set es'
  using sub-rel-of assms(1,2,3)
        extends-tgt-out-edges[OF assms(1)]
        sp-through-de-decomp[of e subs g' v es tgt e]
        sp-ends-in-tgt-imp-mem[of e v es]
  by blast

  moreover
  have subpath g v es' (src e) subs
  proof −
    have v ∈ Graph.vertices g
    using assms(1,3) fst-of-sp-is-vert[OF assms(2)]
    by (auto simp add : vertices-def)

    moreover
```

```
    have SubRel.subpath g' v es' (src e) subs
    using assms(2) ‹es = es' @ [e]› by (simp add : sp-append-one)

    ultimately
    show ?thesis
    using assms(1) sub-rel-of ‹e ∉ set es'›
    unfolding subpath-def by (auto simp add : sub-rel-of-def)
  qed

  ultimately
  show ?thesis by blast
qed

end
theory SubExt
imports SubRel
begin
```

A.11 Extending Subsomption Relations

In this section, we are interested in the evolution of the set of sub-paths of a rooted graph equipped with a subsumption relation after adding a subsumption to this relation. We are only interested in adding subsumptions such that the resulting relation is a well-formed sub-relation of the graph (provided the original relation was such). As for the extension by edges, a number of side conditions must be met for the new subsumption to be added.

A.11.1 Definition

Extending a subsumption relation *subs* consists in adding a subsumption *sub* such that:

- the two vertices involved are distinct,
- they are occurrences of the same vertex,
- they are both vertices of the graph,
- the subsumee must not already be a subsumer or a subsumee,
- the subsumer must not be a subsumee (but it can already be a subsumer),
- the subsumee must have no out-edges.

Once again, in order to ease proofs, we use a predicate stating when a subsumpion relation is the extension of another instead of using a function that would produce the extension.

```
abbreviation extends ::
  (('v × nat),'x) rgraph-scheme ⇒ 'v sub-rel-t ⇒ 'v sub-t ⇒ 'v sub-rel-t ⇒ bool
where
  extends g subs sub subs' ≡ (
     subsumee sub ≠ subsumer sub
   ∧ fst (subsumee sub) = fst (subsumer sub)
   ∧ subsumee sub ∈ Graph.vertices g
   ∧ subsumee sub ∉ subsumers subs
   ∧ subsumee sub ∉ subsumees subs
   ∧ subsumer sub ∈ Graph.vertices g
   ∧ subsumer sub ∉ subsumees subs
   ∧ out-edges g (subsumee sub) = {}
   ∧ subs' = subs ∪ {sub})
```

A.11.2 Properties of extensions

First, we show that such extensions yield sub-relations (resp. well-formed relations), provided the original relation is a sub-relation (resp. well-formed relation).

Extending the sub-relation of a graph yields a new sub-relation for this graph.

```
lemma (in sub-rel-of)
  assumes extends g subs sub subs'
  shows   sub-rel-of g subs'
using assms sub-rel-of unfolding sub-rel-of-def by force
```

Extending a well-formed relation yields a well-formed relation.

```
lemma (in wf-sub-rel) extends-imp-wf-sub-rel :
  assumes extends g subs sub subs'
  shows   wf-sub-rel subs'
unfolding wf-sub-rel-def
proof (intro conjI, goal-cases)
  case 1 show ?case using wf-sub-rel assms by auto
next
  case 2 show ?case
  unfolding Ball-def
  proof (intro allI impI)
    fix v

    assume v ∈ subsumees subs'

    hence  v = subsumee sub ∨ v ∈ subsumees subs using assms by auto

    thus ∃! v'. (v,v') ∈ subs'
    proof (elim disjE, goal-cases)
      case 1 show ?thesis
      unfolding Ex1-def
      proof (rule-tac ?x=subsumer sub in exI, intro conjI)
```

```
      show (v, subsumer sub) ∈ subs' using 1 assms by simp
    next
      have v ∉ subsumees subs using assms 1 by auto

      thus ∀ v'. (v, v') ∈ subs' ⟶ v' = subsumer sub
      using assms by auto force
    qed
  next
    case 2

    then obtain v' where (v,v') ∈ subs by auto

    hence v ≠ subsumee sub
    using assms unfolding subsumees-conv
    by (force simp del : split-paired-All split-paired-Ex)

    show ?thesis
    using assms
          ‹v ≠ subsumee sub›
          ‹(v,v') ∈ subs› subsumed-by-one
    unfolding subsumees-conv Ex1-def
    by (rule-tac ?x=v' in exI)
       (auto simp del : split-paired-All split-paired-Ex)
  qed
 qed
next
 case 3 show ?case using wf-sub-rel assms by auto
qed
```

Thus, extending a well-formed sub-relation yields a well-formed sub-relation.

```
lemma (in wf-sub-rel-of) extends-imp-wf-sub-rel-of :
 assumes extends g subs sub subs'
 shows   wf-sub-rel-of g subs'
using sub-rel-of assms
      wf-sub-rel.extends-imp-wf-sub-rel[OF wf-sub-rel assms]
by (simp add : wf-sub-rel-of-def sub-rel-of-def)
```

A.11.3 Properties of sub-paths in an extension

Extending a sub-relation of a graph preserves the existing sub-paths.

```
lemma sp-in-extends :
 assumes extends g subs sub subs'
 assumes subpath g v1 es v2 subs
 shows   subpath g v1 es v2 subs'
using assms ces-Un[of v1 es v2 subs {sub}]
by (simp add : subpath-def sub-rel-of-def)
```

We want to describe how the addition of a subsumption modifies the

set of sub-paths in the graph. As in the previous theories, we will focus on a small number of theorems expressing sub-paths in extensions as functions of sub-paths in the graphs before extending them (their subsumption relations). We first express sub-paths starting at the subsumee of the new subsumption, then the sub-paths starting at any other vertex.

First, we are interested in sub-paths starting at the subsumee of the new subsumption. Since such vertices have no out-edges, these sub-paths must be either empty or must be sub-paths from the subsumer of this subsumption.

lemma *sp-in-extends-imp1* :
  assumes *extends g subs* (v_1,v_2) *subs'*
  assumes *subpath g* v_1 *es v subs'*
  shows *es* = [] ∨ *subpath g* v_2 *es v subs'*
proof −
  have $(v_1,v_2) \in$ *subs'* **using** *assms(1)* **by** *fast*

  moreover
  hence ∃! *v*. $(v_1,v) \in$ *subs'*
  using *assms(1)* **unfolding** *Ex1-def subsumees-conv* **by** (*rule-tac ?x*=v_2 **in** *exI*) *auto*

  ultimately
  show *?thesis*
  using *assms sp-from-subsumee*[*of* v_1 v_2 *subs' g es v*]
  by *auto*
qed

After an extension, sub-paths starting at any other vertex than the new subsumee are either:

- sub-paths of the graph before the extension if they do not "use" the new subsumption,
- made of a finite number of sub-paths of the graph before the extension if they use the new subsumption.

In order to state the lemmas expressing these facts, we first need to introduce the concept of *usage* of a subsumption by a sub-path.

The idea is that, if a sequence of edges that uses a subsumption *sub* is consistent w.r.t. a subsumption relation *subs*, then *sub* must occur in the transitive closure of *subs* i.e. the consistency of the sequence directly (and partially) depends on *sub*. In the case of well-formed subsumption relations, whose transitive closures equal the relations themselves, the dependency of the consistency reduces to the fact that *sub* is a member of *subs*.

fun *uses-sub* ::
  ('*v* × *nat*) ⇒ ('*v* × *nat*) *edge list* ⇒ ('*v* × *nat*) ⇒ (('*v* × *nat*) × ('*v* × *nat*)) ⇒ *bool*

where
uses-sub v_1 [] v_2 *sub* $= (v_1 \neq v_2 \wedge sub = (v_1,v_2))$
| *uses-sub* v_1 $(e\#es)$ v_2 *sub* $= (v_1 \neq src\ e \wedge sub = (v_1,src\ e) \vee uses\text{-}sub\ (tgt\ e)$
$es\ v_2\ sub)$

In order for a sequence *es* using the subsumption *sub* to be consistent w.r.t. to a subsumption relation *subs*, the subsumption *sub* must occur in the transitive closure of *subs*.

lemma
assumes *uses-sub* v_1 *es* v_2 *sub*
assumes *ces* v_1 *es* v_2 *subs*
shows $sub \in subs^+$
using *assms* **by** (*induction es arbitrary* : v_1) *fastforce+*

This reduces to the membership of *sub* to *subs* when the latter is well-formed.

lemma (**in** *wf-sub-rel*)
assumes *uses-sub* v_1 *es* v_2 *sub*
assumes *ces* v_1 *es* v_2 *subs*
shows $sub \in subs$
using *assms trancl-eq* **by** (*induction es arbitrary* : v_1) *fastforce+*

Sub-paths prior to the extension do not use the new subsumption.

lemma *extends-and-sp-imp-not-using-sub* :
assumes *extends g subs* (v,v') *subs'*
assumes *subpath g* v_1 *es* v_2 *subs*
shows $\neg$ *uses-sub* v_1 *es* v_2 (v,v')
proof (*intro notI*)
assume *uses-sub* v_1 *es* v_2 (v,v')

moreover
have *ces* v_1 *es* v_2 *subs* **using** *assms*(*2*) **by** (*simp add* : *subpath-def*)

ultimately
have $(v,v') \in subs^+$ **by** (*induction es arbitrary* : v_1) *fastforce+*

thus *False*
using *assms*(*1*) **unfolding** *subsumees-conv*
by (*elim conjE*) (*frule tranclD*, *force*)
qed

An intermediate lemma that we will sometimes need to simplify the transitive closure of the extended relation.

lemma *extends-trancl* :
assumes *extends g subs sub subs'*
shows $subs'^+ = subs^+ \cup \{sub\}$
proof −

obtain v_1 v_2 **where** *A*: $sub = (v_1,v_2)$

```
  using surjective-pairing by blast

  moreover
  hence subs'⁺ = subs⁺ ∪ {(v,v'). (v,v1) ∈ subs* ∧ (v2,v') ∈ subs*}
  using assms by (auto simp add : trancl-insert)

  moreover
  have {(v,v'). (v,v1) ∈ subs* ∧ (v2,v') ∈ subs*} = {(v1,v2)}
  unfolding set-eq-iff mem-Collect-eq
  apply (subst (1) split-paired-All)
  unfolding prod.case insert-iff empty-iff prod.inject
  unfolding simp-thms(31)
  apply (intro allI iffI conjI)
  proof (elim conjE, goal-cases)
    case (1 a b) thus ?case
    using assms A rtranclD[of a v1 subs] tranclD2[of a v1 subs]
    unfolding subsumers-conv by auto
  next
    case (2 a b) thus ?case
    using assms A rtranclD[of v2 b subs] tranclD [of v2 b subs]
    unfolding subsumees-conv by auto
  next
    case 3 thus ?case by simp
  next
    case 4 thus ?case by simp
  qed

  ultimately
  show ?thesis by auto
qed
```

Suppose that the empty sequence is a sub-path leading from v_1 to v_2 after the extension. Then, it is also a sub-path leading from v_1 to v_2 in the graph before the extension if and only if (v_1, v_2) is not the new subsumption.

```
lemma sp-Nil-in-extends-imp :
  assumes extends g subs (v,v') subs'
  assumes subpath g v1 [] v2 subs'
  shows   subpath g v1 [] v2 subs ⟷ (v1 ≠ v ∨ v2 ≠ v')
proof (intro iffI, goal-cases)
  case 1 thus ?case
  using assms(1)
        extends-and-sp-imp-not-using-sub[OF assms(1), of v1 [] v2]
  by auto
next
  case 2

  have v1 = v2 ∨ (v1,v2) ∈ subs'⁺
  and  v1 ∈ Graph.vertices g
  using assms(2)
```

```
  by (simp-all add : Nil-sp)

  hence v₁ = v₂ ∨ (v₁,v₂) ∈ subs⁺
  using 2 extends-trancl[OF assms(1)] by auto

  moreover
  have v₂ ∈ Graph.vertices g
  using assms(2) by (intro lst-of-sp-is-vert)

  ultimately
  show subpath g v₁ [] v₂ subs
  using assms by (auto simp add : subpath-def sub-rel-of-def)
qed
```

Thus, sub-paths after the extension that do not use the new subsumption are also sub-paths before the extension.

```
lemma sp-in-extends-not-using-sub :
  assumes extends g subs (v,v') subs'
  assumes subpath g v₁ es v₂ subs'
  assumes ¬ uses-sub v₁ es v₂ (v,v')
  shows   subpath g v₁ es v₂ subs
using assms(2,3)
proof (induction es arbitrary : v₁)
  case Nil thus ?case using assms(1) by (auto simp add : sp-Nil-in-extends-imp)
next
  case (Cons e es')

  hence SubRel.subpath g (tgt e) es' v₂ subs'
  by (simp add : sp-Cons)

  moreover
  have ¬ uses-sub (tgt e) es' v₂ (v, v') using Cons by auto

  ultimately
  have subpath g (tgt e) es' v₂ subs using Cons(1) by fast

  moreover
  have subpath g v₁ [e] (tgt e) subs
  proof −
    have subpath g v₁ [e] (tgt e) subs'
    using Cons(2) by (auto simp add: sp-Cons Nil-sp fst-of-sp-is-vert)

    moreover
    have ¬ uses-sub v₁ [e] (tgt e) (v, v')
    using Cons(3) by auto

    ultimately
    show ?thesis using assms(1) extends-trancl[OF assms(1)]
    by (auto simp add: sp-one sub-rel-of-def)
```

```
  qed

  ultimately
  show ?case by (simp add: sp-Cons)
qed
```

We are finally able to describe sub-paths starting at any other vertex than the new subsumee after the extension. Such sub-paths are made of a finite number of sub-paths before the extension: the usage of the new subsumption between such (sub-)sub-paths makes them sub-paths after the extension. We express this idea as follows. Sub-paths starting at any other vertex than the new subsumee are either:

- sub-paths of the graph before the extension,
- made of a non-empty prefix that is a sub-path leading to the new subsumee in the original graph and a (potentially empty) suffix that is a sub-path starting at the new subsumer after the extension.

For the second case, the lemma `sp_in_extends_imp1` as well as the following lemma could be applied to the suffix in order to decompose it into sub-paths of the graph before extension (combined with the fact that we only consider finite sub-paths, we indirectly obtain that sub-paths after the extension are made of a finite number of sub-paths before the extension, that are made consistent with the new relation by using the new subsumption).

```
lemma sp-in-extends-imp2 :
  assumes extends g subs (v,v') subs'
  assumes subpath g v1 es v2 subs'
  assumes v1 ≠ v

  shows   subpath g v1 es v2 subs ∨ (∃ es1 es2. es = es1 @ es2
                                     ∧ es1 ≠ []
                                     ∧ subpath g v1 es1 v subs
                                     ∧ subpath g v es2 v2 subs')
        (is ?P es v1)
proof (case-tac uses-sub v1 es v2 (v,v'), goal-cases)
  case 1

  thus ?thesis
  using assms(2,3)
  proof (induction es arbitrary : v1)
    case (Nil v1) thus ?case by auto
  next
    case (Cons edge es v1)

    hence v1 = src edge ∨ (v1, src edge) ∈ subs'⁺
    and   edge ∈ edges g
    and   subpath g (tgt edge) es v2 subs'
```

```
  using assms(1)
  by (simp-all add : sp-Cons)

  hence subpath g v1 [edge] (tgt edge) subs'
  using  sp-one by (simp add : subpath-def) fast

  have subpath g v1 [edge] (tgt edge) subs
  proof −
    have ¬ uses-sub v1 [edge] (tgt edge) (v,v')
    using assms(1) Cons(2,4) by auto

    thus ?thesis
    using assms(1) ‹subpath g v1 [edge] (tgt edge) subs'›
    by (elim sp-in-extends-not-using-sub)
  qed

  thus ?case
  proof (case-tac tgt edge = v, goal-cases)
    case 1 thus ?thesis
    using ‹subpath g v1 [edge] (tgt edge) subs›
          ‹subpath g (tgt edge) es v2 subs'›
    by (intro disjI2, rule-tac ?x=[edge] in exI) auto
  next
    case 2

    moreover
    have uses-sub (tgt edge) es v2 (v,v') using Cons(2,4) by simp

    ultimately
    have ?P es (tgt edge)
    using ‹subpath g (tgt edge) es v2 subs'›
    by (intro Cons.IH)

    thus ?thesis
    proof (elim disjE exE conjE, goal-cases)
      case 1 thus ?thesis
      using ‹subpath g (tgt edge) es v2 subs'›
            ‹uses-sub (tgt edge) es v2 (v,v')›
            extends-and-sp-imp-not-using-sub[OF assms(1)]
      by fast
    next
      case (2 es1 es2) thus ?thesis
      using ‹es = es1 @ es2›
            ‹subpath g v1 [edge] (tgt edge) subs›
            ‹subpath g v es2 v2 subs'›
     by (intro disjI2, rule-tac ?x=edge # es1 in exI) (auto simp add : sp-Cons)
    qed
  qed
qed
```

```
next
  case 2 thus ?thesis
  using assms(1,2) by (simp add : sp-in-extends-not-using-sub)
qed

end
theory RB
imports LTS ArcExt SubExt
begin
```

A.12 Red-Black Graphs

In this section we define red-black graphs and the five operators that perform over them. Then, we state and prove a number of intermediate lemmas about red-black graphs built using only these five operators, in other words: invariants about our method of transformation of red-black graphs.

Then, we define the notion of red-black paths and state and prove the main properties of our method, namely its correctness and the fact that it preserves the set of feasible paths of the program under analysis.

A.12.1 Basic definitions

The type of red-black graphs

We represent red-black graph with the following record. We detail its fields:

- *red* is the red graph, called *red part*, which represents the unfolding of the black part. Its vertices are indexed black vertices,
- *black* is the original LTS, the *black part*,
- *subs* is the subsumption relation over the vertices of *red*,
- *init-conf* is the initial configuration,
- *confs* is a function associating configurations to the vertices of *red*,
- *marked* is a function associating truth values to the vertices of *red*. We use it to represent the fact that a particular configuration (associated to a red location) is known to be unsatisfiable,
- *strengthenings* is a function associating boolean expressions over program variables to vertices of the red graph. Those boolean expressions can be seen as invariants that the configuration associated to the "strengthened" red vertex has to model.

We are only interested by red-black graphs obtained by the inductive relation *RedBlack*. From now on, we call "red-black graphs" the *pre-RedBlack*'s obtained by *RedBlack* and "pre-red-black graphs" all other ones.

record ('*vert*,'*var*,'*d*) *pre-RedBlack* =
  red :: ('*vert* × *nat*) *rgraph*
  black :: ('*vert*,'*var*,'*d*) *lts*
  subs :: '*vert sub-rel-t*
  init-conf :: ('*var*,'*d*) *conf*
  confs :: ('*vert* × *nat*) ⇒ ('*var*,'*d*) *conf*
  marked :: ('*vert* × *nat*) ⇒ *bool*
  strengthenings :: ('*vert* × *nat*) ⇒ ('*var*,'*d*) *bexp*

We call *red vertices* the set of vertices of the red graph.

abbreviation *red-vertices* ::
  ('*vert*,'*var*,'*d*,'*x*) *pre-RedBlack-scheme* ⇒ ('*vert* × *nat*) *set*
where
  red-vertices lts ≡ *Graph.vertices* (*red lts*)

ui-edge is the operation of "unindexing" the ends of a red edge, thus giving the corresponding black edge.

abbreviation *ui-edge* ::
  ('*vert* × *nat*) *edge* ⇒ '*vert edge*
where
  ui-edge e ≡ (| *src* = *fst* (*src e*), *tgt* = *fst* (*tgt e*) |)

We extend this idea to sequences of edges.

abbreviation *ui-es* ::
  ('*vert* × *nat*) *edge list* ⇒ '*vert edge list*
where
  ui-es es ≡ *map ui-edge es*

Finite red-black graphs

locale *pre-RedBlack* =
  fixes *prb* :: ('*vert*,'*var*,'*d*) *pre-RedBlack*

We say that a pre-red-black graph is finite if :

- the path predicate of its initial configuration contains a finite number of constraints,
- each of these constraints contains a finite number of variables,
- its black part is finite (cf. definition of *finite-lts*.).

locale *finite-RedBlack* = *pre-RedBlack* +
  assumes *finite-init-pred* : *finite* (*pred* (*init-conf prb*))
  assumes *finite-init-pred-symvars* : ∀ *e* ∈ *pred* (*init-conf prb*). *finite* (*Bexp.vars e*)
  assumes *finite-lts* : *finite-lts* (*black prb*)
begin
  lemmas *finite-RedBlack* = *finite-init-pred finite-init-pred-symvars finite-lts*
end

A.12.2 Extensions of red-black graphs

We now define the five basic operations that can be performed over red-black graphs. Since we do not want to model the heuristics part of our prototype, a number of conditions must be met for each operator to apply. For example, in our prototype abstractions are performed at nodes that actually have successors, and these abstractions must be propagated to these successors in order to keep the symbolic execution graph consistent. Propagation is a complex task, and it is hard to model in Isabelle/HOL. This is partially due to the fact that we model the red part as a graph, in which propagation might not terminate. Instead, we suppose that abstraction must be performed only at leaves of the red part. This is equivalent to implicitly assume the existence of an oracle that would tell that we will need to abstract some red vertex and how to abstract it, as soon as this red vertex is added to the red part.

As in the previous theories, we use predicates instead of functions to model these transformations to ease writing and reading definitions, proofs, etc.

Extension by symbolic execution

The core abstract operation of symbolic execution: take a black edge and turn it red, by symbolic execution of its label. In the following abbreviation, *re* is the red edge obtained from the (hypothetical) black edge *e* that we want to symbolically execute and *c* the configuration obtained by symbolic execution of the label of *e*. Note that this extension could have been defined as a predicate that takes only two *pre-RedBlack*s and evaluates to *true* if and only if the second has been obtained by adding a red edge as a result of symbolic execution. However, making the red edge and the configuration explicit allows for lighter definitions, lemmas and proofs in the following.

abbreviation *se-extends* ::
  ('vert,'var,'d) pre-RedBlack ⇒
  ('vert × nat) edge ⇒
  ('var,'d) conf ⇒
  ('vert,'var,'d) pre-RedBlack ⇒ bool
where
  se-extends prb re c prb' ≡
    ui-edge re ∈ edges (black prb)
  ∧ ArcExt.extends (red prb) re (red prb')
  ∧ src re ∉ subsumees (subs prb)
  ∧ SE (confs prb (src re)) (labeling (black prb) (ui-edge re)) c
  ∧ prb' = (| red = red prb',
        black = black prb,
        subs = subs prb,
        init-conf = init-conf prb,
        confs = (confs prb) (tgt re := c),
        marked = (marked prb)(tgt re := marked prb (src re)),

strengthenings = *strengthenings prb* ⦈

Hiding the new red edge (using an existential quantifier) and the new configuration makes the following abbreviation more intuitive. However, this would require using `obtain` or `let ... = ... in ...` constructs in the following lemmas and proofs, making them harder to read and write.

abbreviation *se-extends2* ::
$('vert,'var,'d)$ *pre-RedBlack* ⇒ $('vert,'var,'d)$ *pre-RedBlack* ⇒ *bool*
where
se-extends2 prb prb' ≡
∃ *re* ∈ *edges* (*red prb'*).
ui-edge re ∈ *edges* (*black prb*)
∧ *ArcExt.extends* (*red prb*) *re* (*red prb'*)
∧ *src re* ∉ *subsumees* (*subs prb*)
∧ *SE* (*confs prb* (*src re*)) (*labeling* (*black prb*) (*ui-edge re*)) (*confs prb'* (*tgt re*))
∧ *black prb'* = *black prb*
∧ *subs prb'* = *subs prb*
∧ *init-conf prb'* = *init-conf prb*
∧ *confs prb'* = (*confs prb*) (*tgt re* := *confs prb'* (*tgt re*))
∧ *marked prb'* = (*marked prb*)(*tgt re* := *marked prb* (*src re*))
∧ *strengthenings prb'* = *strengthenings prb*

Extension by marking

The abstract operation of mark-as-unsat. It manages the information - provided, for example, by an external automated prover, that the configuration of the red vertex *rv* has been proved unsatisfiable.

abbreviation *mark-extends* ::
$('vert,'var,'d)$ *pre-RedBlack* ⇒ $('vert \times nat)$ ⇒ $('vert,'var,'d)$ *pre-RedBlack* ⇒ *bool*
where
mark-extends prb rv prb' ≡
rv ∈ *red-vertices prb*
∧ *out-edges* (*red prb*) *rv* = {}
∧ *rv* ∉ *subsumees* (*subs prb*)
∧ *rv* ∉ *subsumers* (*subs prb*)
∧ ¬ *sat* (*confs prb rv*)
∧ *prb'* = ⦇ *red* = *red prb*,
black = *black prb*,
subs = *subs prb*,
init-conf = *init-conf prb*,
confs = *confs prb*,
marked = (λ *rv'*. *if rv'* = *rv then True else marked prb rv'*),
strengthenings = *strengthenings prb* ⦈

Extension by subsumption

The abstract operation of introducing a subsumption link.

abbreviation *subsum-extends* ::
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *'vert sub-t* $\Rightarrow$ $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
subsum-extends prb sub prb' $\equiv$
SubExt.extends (*red prb*) (*subs prb*) *sub* (*subs prb'*)
$\wedge$ $\neg$ *marked prb* (*subsumer sub*)
$\wedge$ $\neg$ *marked prb* (*subsumee sub*)
$\wedge$ *confs prb* (*subsumee sub*) $\sqsubseteq$ *confs prb* (*subsumer sub*)
$\wedge$ *prb'* = (| *red* = *red prb*,
black = *black prb*,
subs = *insert sub* (*subs prb*),
init-conf = *init-conf prb*,
confs = *confs prb*,
marked = *marked prb*,
strengthenings = *strengthenings prb*,
... = *more prb* |)

Extension by abstraction

This operation replaces the configuration of a red vertex *rv* by an abstraction of this configuration. The way the abstraction is computed is not specified. However, besides a number of side conditions, it must subsume the former configuration of *rv* and must entail its safeguard condition, if any.

abbreviation *abstract-extends* ::
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
$('vert \times nat)$ $\Rightarrow$
$('var,'d)$ *conf* $\Rightarrow$
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
abstract-extends prb rv c_a *prb'* $\equiv$
rv $\in$ *red-vertices prb*
$\wedge$ $\neg$ *marked prb rv*
$\wedge$ *out-edges* (*red prb*) *rv* = {}
$\wedge$ *rv* $\notin$ *subsumees* (*subs prb*)
$\wedge$ *abstract* (*confs prb rv*) c_a
$\wedge$ $c_a \models_c$ (*strengthenings prb rv*)
$\wedge$ *finite* (*pred* c_a)
$\wedge$ ($\forall$ *e* $\in$ *pred* c_a. *finite* (*vars e*))
$\wedge$ *prb'* = (| *red* = *red prb*,
black = *black prb*,
subs = *subs prb*,
init-conf = *init-conf prb*,
confs = (*confs prb*)(*rv* := c_a),
marked = *marked prb*,
strengthenings = *strengthenings prb*,
... = *more prb* |)

Extension by strengthening

This operation consists in labeling a red vertex with a safeguard condition. It does not actually change the red part, but model the mechanism of preventing too crude abstractions.

abbreviation *strengthen-extends* ::
 $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$
 $('vert \times nat)$ $\Rightarrow$
 $('var,'d)$ *bexp* $\Rightarrow$
 $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
 strengthen-extends prb rv e prb' $\equiv$
  $rv \in$ *red-vertices prb*
 $\wedge$ $rv \notin$ *subsumees* (*subs prb*)
 $\wedge$ *confs prb rv* $\models_c$ *e*
 $\wedge$ *prb'* = (| *red* = *red prb*,
  black = *black prb*,
  subs = *subs prb*,
  init-conf = *init-conf prb*,
  confs = *confs prb*,
  marked = *marked prb*,
  strengthenings = (*strengthenings prb*)(*rv* := (λ σ. (*strengthenings prb rv*) σ $\wedge$ *e* σ)),
  ... = *more prb* |)

A.12.3 Building red-black graphs using extensions

Red-black graphs are pre-red-black graphs built with the following inductive relation, i.e. using only the five previous pre-red-black graphs transformation operators, starting from an empty red part.

inductive *RedBlack* ::
 $('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
 base :
  fst (*root* (*red prb*)) = *init* (*black prb*) $\Longrightarrow$
  edges (*red prb*) = {} $\Longrightarrow$
  subs prb = {} $\Longrightarrow$
  (*confs prb*) (*root* (*red prb*)) = *init-conf prb* $\Longrightarrow$
  marked prb = (λ *rv*. *False*) $\Longrightarrow$
  strengthenings prb = (λ *rv*. (λ σ. *True*)) $\Longrightarrow$ *RedBlack prb*

| *se-step* :
  RedBlack prb $\Longrightarrow$
  se-extends prb re c prb' $\Longrightarrow$ *RedBlack prb'*

| *mark-step* :
  RedBlack prb $\Longrightarrow$
  mark-extends prb rv prb' $\Longrightarrow$ *RedBlack prb'*

| *subsum-step* :
  RedBlack prb $\Longrightarrow$
  subsum-extends prb sub prb' $\Longrightarrow$ *RedBlack prb'*

| *abstract-step* :
  RedBlack prb $\Longrightarrow$
  abstract-extends prb rv c_a *prb'* $\Longrightarrow$ *RedBlack prb'*

| *strengthen-step* :
  RedBlack prb $\Longrightarrow$
  strengthen-extends prb rv e prb' $\Longrightarrow$ *RedBlack prb'*

A.12.4 Properties of red-black graphs

Invariants of red-black graphs

Red edges are specific versions of black edges.

lemma *ui-re-is-be* :
 assumes *RedBlack prb*
 assumes *re* $\in$ *edges* (*red prb*)
 shows *ui-edge re* $\in$ *edges* (*black prb*)
using *assms* **by** (*induct rule* : *RedBlack.induct*) *auto*

The set of out-going edges from a red vertex is a subset of the set of out-going edges from the black location it represents.

lemma *red-OA-subset-black-OA* :
 assumes *RedBlack prb*
 shows *ui-edge* ‘ *out-edges* (*red prb*) *rv* $\subseteq$ *out-edges* (*black prb*) (*fst rv*)
using *assms* **by** (*induct prb*) (*fastforce simp add* : *vertices-def*)+

The red root is an indexed version of the black initial location.

lemma *consistent-roots* :
 assumes *RedBlack prb*
 shows *fst* (*root* (*red prb*)) = *init* (*black prb*)
using *assms* **by** (*induct prb*) *auto*

Red locations of a red-black graph are indexed versions of its black locations.

lemma *ui-rv-is-bv* :
 assumes *RedBlack prb*
 assumes *rv* $\in$ *red-vertices prb*
 shows *fst rv* $\in$ *Graph.vertices* (*black prb*)
using *assms consistent-roots ui-re-is-be*
by (*auto simp add* : *vertices-def image-def Bex-def*) *fastforce*+

The subsumption relation of a red-black graph is a sub-relation of its red part.

lemma *subs-sub-rel-of* :

```
  assumes RedBlack prb
  shows   sub-rel-of (red prb) (subs prb)
using assms unfolding sub-rel-of-def
proof (induct prb)
  case base thus ?case by simp
next
  case se-step thus ?case
  by (elim conjE) (auto simp add : vertices-def)
next
  case mark-step thus ?case by auto
next
  case subsum-step thus ?case by auto
next
  case abstract-step thus ?case by simp
next
  case strengthen-step thus ?case by simp
qed
```

The subsumption relation of red-black graph is well-formed.

```
lemma subs-wf-sub-rel :
  assumes RedBlack prb
  shows   wf-sub-rel (subs prb)
using assms
proof (induct prb)
  case base thus ?case by (simp add : wf-sub-rel-def)
next
  case se-step thus ?case by force
next
  case mark-step thus ?case by (auto simp add : wf-sub-rel-def)
next
  case subsum-step thus ?case
  by (auto simp add : wf-sub-rel.extends-imp-wf-sub-rel)
next
  case abstract-step thus ?case by simp
next
  case strengthen-step thus ?case by simp
qed
```

Using the two previous lemmas, we have that the subsumption relation of a red-black graph is a well-formed sub-relation of its red-part.

```
lemma subs-wf-sub-rel-of :
  assumes RedBlack prb
  shows   wf-sub-rel-of (red prb) (subs prb)
using assms subs-sub-rel-of subs-wf-sub-rel by (simp add : wf-sub-rel-of-def) fast
```

Subsumptions only involve red locations representing the same black location.

```
lemma subs-to-same-BL :
  assumes RedBlack prb
```

assumes *sub* ∈ *subs prb*
shows *fst* (*subsumee sub*) = *fst* (*subsumer sub*)
using *assms subs-wf-sub-rel* **unfolding** *wf-sub-rel-def* **by** *fast*

If a red edge sequence *res* is consistent between red locations *rv1* and *rv2* with respect to the subsumption relation of a red-black graph, then its unindexed version is consistent between the black locations represented by *rv1* and *rv2*.

lemma *rces-imp-bces* :
assumes *RedBlack prb*
assumes *SubRel.ces* rv_1 *res* rv_2 (*subs prb*)
shows *Graph.ces* (*fst* rv_1) (*ui-es res*) (*fst* rv_2)
using *assms*
proof (*induct res arbitrary* : rv_1)
case (*Nil* rv_1) **thus** *?case*
using *wf-sub-rel.trancl-eq*[*OF subs-wf-sub-rel*] *subs-to-same-BL*
by *fastforce*
next
case (*Cons re res* rv_1)

hence *1* : rv_1 = *src re* ∨ (rv_1, *src re*) ∈ $(subs\ prb)^+$
and *2* : *ces* (*tgt re*) *res* rv_2 (*subs prb*) **by** *simp-all*

have *src* (*ui-edge re*) = *fst* rv_1
using *1*
wf-sub-rel.trancl-eq[*OF subs-wf-sub-rel*[*OF assms*(*1*)]]
subs-to-same-BL[*OF assms*(*1*), *of* (rv_1,*src re*)]
by *auto*

moreover
have *Graph.ces* (*tgt* (*ui-edge re*)) (*ui-es res*) (*fst* rv_2)
using *assms*(*1*) *Cons*(*1*) *2* **by** *simp*

ultimately
show *?case* **by** *simp*
qed

The unindexed version of a sub-path in the red part of a red-black graph is a sub-path in its black part. This is an important fact: in the end, it helps proving that set of paths we consider in red-black graphs are paths of the original LTS. Thus, the same states are computed along these paths.

theorem *red-sp-imp-black-sp* :
assumes *RedBlack prb*
assumes *subpath* (*red prb*) rv_1 *res* rv_2 (*subs prb*)
shows *Graph.subpath* (*black prb*) (*fst* rv_1) (*ui-es res*) (*fst* rv_2)
using *assms rces-imp-bces ui-rv-is-bv ui-re-is-be*
unfolding *subpath-def Graph.subpath-def* **by** (*intro conjI*) (*fast*, *fast*, *fastforce*)

Any constraint in the path predicate of a configuration associated to a

red location of a red-black graph contains a finite number of variables.

lemma *finite-pred-constr-symvars* :
  assumes *RedBlack prb*
  assumes *finite-RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  shows $\forall\ e \in$ *pred* (*confs prb rv*). *finite* (*Bexp.vars e*)
using *assms*
proof (*induct prb arbitrary* : *rv*)
  case *base* **thus** *?case* **by** (*simp add* : *vertices-def finite-RedBlack-def*)
next
  case (*se-step prb re c′ prb′*)

  hence $rv \in$ *red-vertices prb* $\vee$ *rv* = *tgt re* **by** (*auto simp add* : *vertices-def*)

  thus *?case*
  proof (*elim disjE*)
    assume $rv \in$ *red-vertices prb*

    moreover
    have *finite-RedBlack prb*
    using *se-step*(*3*,*4*) **by** (*auto simp add* : *finite-RedBlack-def*)

    ultimately
    show *?thesis*
    using *se-step*(*2*,*3*) **by** (*elim conjE*) (*auto simp add* : *vertices-def*)
  next
    assume *rv* = *tgt re*

    moreover
    have *finite-label* (*labeling* (*black prb*) (*ui-edge re*))
    using *se-step* **by** (*auto simp add* : *finite-RedBlack-def*)

    moreover
    have $\forall\ e \in$ *pred* (*confs prb* (*src re*)). *finite* (*Bexp.vars e*)
    using *se-step se-step*(*2*)[*of src re*] **unfolding** *finite-RedBlack-def*
    by (*elim conjE*) *auto*

    moreover
    have *SE* (*confs prb* (*src re*)) (*labeling* (*black prb*) (*ui-edge re*)) *c′*
    using *se-step* **by** *auto*

    ultimately
    show *?thesis* **using** *se-step SE-preserves-finiteness1* **by** *fastforce*
  qed
next
  case *mark-step* **thus** *?case* **by** (*simp add* : *finite-RedBlack-def*)
next
  case *subsum-step* **thus** *?case* **by** (*simp add* : *finite-RedBlack-def*)
next

```
  case abstract-step thus ?case by (auto simp add : finite-RedBlack-def)
next
  case strengthen-step thus ?case by (simp add : finite-RedBlack-def)
qed
```

The path predicate of a configuration associated to a red location of a red-black graph contains a finite number of constraints.

```
lemma finite-pred :
  assumes RedBlack prb
  assumes finite-RedBlack prb
  assumes rv ∈ red-vertices prb
  shows   finite (pred (confs prb rv))
using assms
proof (induct prb arbitrary : rv)
  case base thus ?case by (simp add : vertices-def finite-RedBlack-def)
next
  case (se-step prb re c′ prb′)

  hence rv ∈ red-vertices prb ∨ rv = tgt re
  by (auto simp add : vertices-def)

  thus ?case
  proof (elim disjE, goal-cases)
    case 1 thus ?thesis
    using se-step(2)[of rv] se-step(3,4)
    by (auto simp add : finite-RedBlack-def)
  next
    case 2

    moreover
    hence src re ∈ red-vertices prb
    and  finite (pred (confs prb (src re)))
    using se-step(2)[of src re] se-step(3,4)
    by (auto simp add : finite-RedBlack-def)

    ultimately
    show ?thesis
    using se-step(3) SE-preserves-finiteness2
    by auto
  qed
next
  case mark-step thus ?case by (simp add : finite-RedBlack-def)
next
  case subsum-step thus ?case by (simp add : finite-RedBlack-def)
next
  case abstract-step thus ?case by (simp add : finite-RedBlack-def)
next
  case strengthen-step thus ?case by (simp add : finite-RedBlack-def)
qed
```

Hence, for a red location *rv* of a red-black graph and any label *l*, there exists a configuration that can be obtained by symbolic execution of *l* from the configuration associated to *rv*.

lemma (**in** *finite-RedBlack*) *ex-se-succ* :
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  shows $\exists$ c'. *SE* (*confs prb rv*) *l* c'
using *finite-RedBlack assms*
    finite-imp-ex-SE-succ[*of confs prb rv*]
    finite-pred[*of prb rv*]
    finite-pred-constr-symvars[*of prb rv*]
unfolding *finite-RedBlack-def* **by** *fast*

Generalization of the previous lemma to a list of labels.

lemma (**in** *finite-RedBlack*) *ex-SE-star-succ* :
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  assumes *finite-labels ls*
  shows $\exists$ c'. *SE-star* (*confs prb rv*) *ls* c'
using *finite-RedBlack assms*
    finite-imp-ex-SE-star-succ[*of confs prb rv ls*]
    finite-pred[*OF assms*(*1*), *of rv*]
    finite-pred-constr-symvars[*OF assms*(*1*), *of rv*]
unfolding *finite-RedBlack-def* **by** *simp*

Hence, for any red sub-path, there exists a configuration that can be obtained by symbolic execution of its trace from the configuration associated to its source.

lemma (**in** *finite-RedBlack*) *sp-imp-ex-SE-star-succ* :
  assumes *RedBlack prb*
  assumes *subpath* (*red prb*) rv_1 *res* rv_2 (*subs prb*)
  shows $\exists$ *c*. *SE-star*
        (*confs prb* rv_1)
        (*trace* (*ui-es res*) (*labeling* (*black prb*)))
        c
using *finite-RedBlack assms ex-SE-star-succ*
by (*simp add* : *subpath-def finite-RedBlack-def*)

The configuration associated to a red location *rl* is updatable.

lemma (**in** *finite-RedBlack*)
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  shows *updatable* (*confs prb rv*)
using *finite-RedBlack assms*
    finite-conj[*OF*
      finite-pred[*OF assms*(*1*)]
      finite-pred-constr-symvars[*OF assms*(*1*)]]
    finite-pred-imp-SE-updatable

```
unfolding finite-RedBlack-def by fast
```

The configuration associated to the first member of a subsumption is subsumed by the configuration at its second member.

```
lemma sub-subsumed :
  assumes RedBlack prb
  assumes sub ∈ subs prb
  shows   confs prb (subsumee sub) ⊑ confs prb (subsumer sub)
using assms
proof (induct prb)
  case base thus ?case by simp
next
  case (se-step prb re c' prb')

  moreover
  hence sub ∈ subs prb by auto

  hence subsumee sub ∈ red-vertices prb
  and   subsumer sub ∈ red-vertices prb
  using se-step(1) subs-sub-rel-of
  unfolding sub-rel-of-def by fast+

  moreover
  have tgt re ∉ red-vertices prb using se-step by auto

  ultimately
  show ?case by auto
next
  case mark-step thus ?case by simp
next
  case (subsum-step prb sub prb') thus ?case by auto
next
  case (abstract-step prb rv c_a prb')

  hence rv ≠ subsumee sub by auto

  show ?case
  proof (case-tac rv = subsumer sub)
    assume rv = subsumer sub

    moreover
    hence  confs prb (subsumer sub) ⊑ confs prb' (subsumer sub)
    using abstract-step by (auto simp add : abstract-def)

    ultimately
    show ?thesis
    using abstract-step
          subsums-trans[of confs prb  (subsumee sub)
                           confs prb  (subsumer sub)
```

confs prb′ (subsumer sub)]
by (*simp add* : *subsums-refl*)
next
assume $rv \neq$ *subsumer sub* **thus** *?thesis* **using** *abstract-step* ‹$rv \neq$ *subsumee sub*› **by** *simp*
qed
next
case *strengthen-step* **thus** *?case* **by** *simp*
qed

Simplification lemmas for sub-paths of the red part.

lemma *rb-Nil-sp* :
assumes *RedBlack prb*
shows *subpath (red prb)* rv_1 [] rv_2 *(subs prb)* =
($rv_1 \in$ *red-vertices prb* $\wedge$ ($rv_1 = rv_2 \vee (rv_1,rv_2) \in$ (*subs prb*)))
using *assms subs-wf-sub-rel subs-sub-rel-of wf-sub-rel.Nil-sp* **by** *fast*

lemma *rb-sp-one* :
assumes *RedBlack prb*
shows *subpath (red prb)* rv_1 [*re*] rv_2 *(subs prb)* =
(*sub-rel-of (red prb) (subs prb)*
$\wedge$ (rv_1 = *src re* $\vee$ (rv_1, *src re*) $\in$ (*subs prb*))
$\wedge$ *re* $\in$ *edges (red prb)* $\wedge$ (*tgt re* = rv_2 $\vee$ (*tgt re*, rv_2) $\in$ (*subs prb*)))
using *assms subs-wf-sub-rel wf-sub-rel.sp-one* **by** *fast*

lemma *rb-sp-Cons* :
assumes *RedBlack prb*
shows *subpath (red prb)* rv_1 (*re* # *res*) rv_2 *(subs prb)* =
(*sub-rel-of (red prb) (subs prb)*
$\wedge$ (rv_1 = *src re* $\vee$ (rv_1, *src re*) $\in$ *subs prb*)
$\wedge$ *re* $\in$ *edges (red prb)*
$\wedge$ *subpath (red prb) (tgt re) res* rv_2 *(subs prb)*)
using *assms subs-wf-sub-rel wf-sub-rel.sp-Cons* **by** *fast*

lemma *rb-sp-append-one* :
assumes *RedBlack prb*
shows *subpath (red prb)* rv_1 (*res* @ [*re*]) rv_2 *(subs prb)* =
(*subpath (red prb)* rv_1 *res (src re) (subs prb)*
$\wedge$ *re* $\in$ *edges (red prb)*
$\wedge$ (*tgt re* = rv_2 $\vee$ (*tgt re*, rv_2) $\in$ *subs prb*))
using *assms subs-wf-sub-rel wf-sub-rel.sp-append-one sp-append-one* **by** *fast*

A.12.5 Relation between red-vertices

The following key-theorem describes the relation between two red locations that are linked by a red sub-path. In a classical symbolic execution tree, the configuration at the end should be the result of symbolic execution of the trace of the sub-path from the configuration at its source. Here, due to the facts that abstractions might have occurred and that we consider sub-paths going through subsumption links, the configuration at the end subsumes the configuration one would obtain by symbolic execution of the trace. Note however that this is only true for configurations computed during the analysis: concrete execution of the sub-paths would yield the same program states than their counterparts in the original LTS.

theorem (**in** *finite-RedBlack*) *SE-rel* :
  assumes *RedBlack prb*
  assumes *subpath* (*red prb*) rv_1 *res* rv_2 (*subs prb*)
  assumes *SE-star* (*confs prb* rv_1) (*trace* (*ui-es res*) (*labeling* (*black prb*))) *c*
  shows $c \sqsubseteq$ (*confs prb* rv_2)
using *assms finite-RedBlack*
find-theorems *name*:*RedBlack*. *name*:*induct*
proof (*induct arbitrary* : rv_1 *res* rv_2 *c rule* : *RedBlack.induct*)

— If the red part is empty, then $rv_1 = rv_2$ and *confs prb* rv_1 = *confs prb* rv_2 which prooves the goal, subsumption being reflexive.

  case (*base prb* rv_1 *res* rv_2 *c*) **thus** *?case*
  by (*force simp add* : *subpath-def Nil-sp subsums-refl*)

next
  — We split the goal into four cases:

- rv_1 and rv_2 are vertices of the old red part,
- rv_1 is a vertex in the old red part, rv_2 is the target of the new edge *re*,
- rv_1 is the target of *re*, rv_2 is a vertex of the old red part,
- rv_1 are rv_2 both equal to the target of *re*.

  case (*se-step prb re c' prb'* rv_1 *res* rv_2 *c*)

  have $rv_1 \in$ *red-vertices prb'*
  and $rv_2 \in$ *red-vertices prb'*
  using *fst-of-sp-is-vert*[*OF se-step*(*4*)]
      lst-of-sp-is-vert[*OF se-step*(*4*)]
  by *simp-all*

  hence $rv_1 \in$ *red-vertices prb* $\wedge$ $rv_1 \neq$ *tgt re* $\vee$ rv_1 = *tgt re*
  and $rv_2 \in$ *red-vertices prb* $\wedge$ $rv_2 \neq$ *tgt re* $\vee$ rv_2 = *tgt re*
  using *se-step* **by** (*auto simp add* : *vertices-def*)

thus *?case*
proof (*elim disjE conjE*, *goal-cases*)
— Both rv_1 and rv_2 are vertices of the old red part.
case *1*

— Hence *res* is also a subpath from rv_1 to rv_2 in the old red part.
moreover
hence *subpath* (*red prb*) rv_1 *res* rv_2 (*subs prb*)
using *se-step*(*1*,*3*,*4*)
sub-rel-of.sp-from-old-verts-imp-sp-in-old
[*OF subs-sub-rel-of*, *of prb re red prb'* rv_1 rv_2 *res*]
by *auto*

— Thus, we use the induction hypothesis (IH) to conclude.
ultimately
show *?thesis* **using** *se-step*
by (*fastforce simp add* : *finite-RedBlack-def*)

next
— rv_1 is a vertex of the old red part, rv_2 is the target of *re*.
case *2*

— Hence *res* is empty or *re* occurs only one time in *res* : at its end.
hence $\exists$ *res'*. *res* = *res'* @ [*re*]
$\wedge$ *re* $\notin$ *set res'*
$\wedge$ *subpath* (*red prb*) rv_1 *res'* (*src re*) (*subs prb*)
using *se-step*
sub-rel-of.sp-to-new-edge-tgt-imp
[*OF subs-sub-rel-of*, *of prb re red prb'* rv_1 *res*]
by *auto*

thus *?thesis*
proof (*elim exE conjE*)
— If *res* = *res'* @ [*re*], then there exists a configuration c' such that :

- c' is obtained from *confs prb* rv_1 by symbolic execution of (the trace of) *res*,
- c' is subsumed by *confs prb* (*src re*) (by IH),
- c is obtained from c' by symbolic execution of *re*.

Moreover, we have that *confs prb* rv_2 is obtained from *confs prb* (*src re*) by symbolic execution of *re*.
Ultimately, we proof the goal by monotonicity of symbolic execution wrt subsumption.
fix *res'*

assume *res* = *res'* @ [*re*]
and *re* $\notin$ *set res'*
and *subpath* (*red prb*) rv_1 *res'* (*src re*) (*subs prb*)

moreover

```
    then obtain c'
    where SE-star (confs prb rv1) (trace (ui-es res') (labeling (black prb))) c'
    and   SE c' (labeling (black prb) (ui-edge re)) c
    using se-step 2 SE-star-append-one by auto blast

    ultimately
    have c' ⊑ (confs prb (src re)) using se-step by fastforce

    thus ?thesis
    using se-step ‹rv1 ≠ tgt re› 2
          ‹SE c' (labeling (black prb) (ui-edge re)) c›
    by (auto simp add : SE-mono-for-sub)
  qed
next
```

— rv_1 is the target of *re*. Hence res is empty and rv_2 also equals *tgt re*, which contradicts our hypothesis.

```
  case 3

  moreover
  have rv1 = rv2
  proof −
    have (rv1,rv2) ∈ (subs prb')
    using se-step 3
          sub-rel-of.sp-from-tgt-in-extends-is-Nil
          [OF subs-sub-rel-of[OF se-step(1)], of re red prb' res rv2]
          rb-Nil-sp[OF RedBlack.se-step[OF se-step(1,3)], of rv1 rv2]
    by auto

    hence rv1 ∈ subsumees (subs prb) using se-step(3) by force

    thus ?thesis
    using se-step ‹rv1 = tgt re› subs-sub-rel-of[OF se-step(1)]
    by (auto simp add  : sub-rel-of-def)
  qed

  ultimately
  show ?thesis by simp

next
```

— Finally, if rv_1 and rv_2 both equal *tgt re*, then we conclude using the fact that the subsumption is reflexive.

```
  case 4

  moreover
  hence res = []
  using se-step
        sub-rel-of.sp-from-tgt-in-extends-is-Nil
        [OF subs-sub-rel-of[OF se-step(1)], of re red prb' res rv2]
  by auto
```

ultimately
show *?thesis* **using** *se-step* **by** (*simp add* : *subsums-refl*)
qed

next
— Marking a red vertex does not affect the configurations associated to red vertices, hence this case is trivial when observing that a subpath after marking is a subpath before marking (which allows to apply the IH).
case (*mark-step prb rv prb'*) **thus** *?case* **by** *simp*

next
case (*subsum-step prb sub prb'* rv_1 *res* rv_2 *c*)

— The fact that *prb'* is also red-black will be needed several times in the following.

have *RB'* : *RedBlack prb'* **by** (*rule RedBlack.subsum-step*[*OF subsum-step*(*1*,*3*)])

— First, we suppose that *res* starts at the newly subsumed red vertex.
show *?case*
proof (*case-tac* rv_1 = *subsumee sub*)

— In this case, *res* is either empty, or a subpath starting at the subsumer of the new subsumption.

assume rv_1 = *subsumee sub*

hence *res* = [] ∨ *subpath* (*red prb'*) (*subsumer sub*) *res* rv_2 (*subs prb'*)
using *subsum-step*(*3*,*4*)
sp-in-extends-imp1[*of subsumee sub subsumer sub red prb subs prb*]
by *simp*

thus *?thesis*
proof (*elim disjE*)
— If *res* is empty, then rv_1 equals rv_2 or (rv_1, rv_2) is in the new subsumption relation.
assume *res* = []

hence $rv_1 = rv_2 \vee (rv_1,rv_2) \in$ (*subs prb'*)
using *subsum-step rb-Nil-sp*[*OF RB'*] **by** *fast*

thus *?thesis*
proof (*elim disjE*)
— If $rv_1 = rv_2$, their configurations are also equal. Moreover, *res* being empty, *c* is the configuration at rv_1. We conclude using reflexivity of subsumption.

assume $rv_1 = rv_2$ **thus** *?thesis*
using *subsum-step*(*5*) ‹*res* = []›
by (*simp add* : *subsums-refl*)

next

— If $(rv_1,\ rv_2)$ is in the new subsumption relation, then the configuration at rv_1 is subsumed by the configuration at rv_2. We conclude using the fact c is the configuration at rv_1.

assume $(rv_1,\ rv_2) \in (subs\ prb')$
thus *?thesis*
using *subsum-step(5)* ‹*res* = []›
sub-subsumed[*OF RB'*, *of* (rv_1,rv_2)]
by *simp*
qed

next

— If *res* is also a subpath from the subsumer of the new subsumption, we show the goal by (backward) induction on *res*.

assume *subpath* (*red prb'*) (*subsumer sub*) *res* rv_2 (*subs prb'*)

thus *?thesis*
using *subsum-step(5)*
proof (*induct res arbitrary* : rv_2 *c rule* : *rev-induct*, *goal-cases*)

— If the red subpath is empty, then $(rv_1,\ rv_2)$ is the new subsumption, which gives the goal by definition of *λprb sub prb'*. (*subsumee sub* ≠ *subsumer sub* ∧ *fst* (*subsumee sub*) = *fst* (*subsumer sub*) ∧ *subsumee sub* ∈ *red-vertices prb* ∧ *subsumee sub* ∉ *subsumers* (*subs prb*) ∧ *subsumee sub* ∉ *subsumees* (*subs prb*) ∧ *subsumer sub* ∈ *red-vertices prb* ∧ *subsumer sub* ∉ *subsumees* (*subs prb*) ∧ *out-edges* (*red prb*) (*subsumee sub*) = {} ∧ *subs prb'* = *subs prb* ∪ {*sub*}) ∧ ¬ *marked prb* (*subsumer sub*) ∧ ¬ *marked prb* (*subsumee sub*) ∧ *confs prb* (*subsumee sub*) ⊑ *confs prb* (*subsumer sub*) ∧ *prb'* = (|*red* = *red prb*, *black* = *black prb*, *subs* = *insert sub* (*subs prb*), *init-conf* = *init-conf prb*, *confs* = *confs prb*, *marked* = *marked prb*, *strengthenings* = *strengthenings prb*, ... = *pre-RedBlack.more prb*|).

case (*1* rv_2 *c*)

have rv_2 = *subsumer sub*
proof −
have (*subsumer sub*,rv_2) ∉ *subs prb'*
proof (*intro notI*)
assume (*subsumer sub*,rv_2) ∈ *subs prb'*

hence *subsumer sub* ∈ *subsumees* (*subs prb'*) **by** *force*

moreover
have *subsumer sub* ∈ *subsumers* (*subs prb'*)
using *subsum-step(3)* **by** *force*

ultimately
show *False*
using *subs-wf-sub-rel*[*OF RB'*]
unfolding *wf-sub-rel-def*
by *auto*
qed

thus *?thesis* **using** *1(1) rb-Nil-sp[OF RB']* **by** *auto*
qed

thus *?case*
using *subsum-step(3) 1(2)* ‹rv_1 = *subsumee sub*› **by** *simp*

next
— Inductive case : the red subpath has the form *res* @ [*re*].
case (*2 re res* rv_2 *c*)

— We call :

- c' the configuration obtained by symbolic execution of *res* from the configuration at rv_1,
- c'' the configuration obtained by symbolic execution of *re* from the configuration at the source of *re*.

We show that c' is subsumed by the configuration at the source of *re* (using "internal" IH), hence *c* is subsumed by c'', by monotonicity of symbolic execution for subsumption.
Moreover, we show that c'' is subsumed by the configuration at the target of *re*, using the fact that [*re*] is a subpath from the source of *re* to its target in the old red part with the "external" IH.
Finally, we show that the configuration at the target of *re* is subsumed by the configuration at rv_2 by observing that the target of *re* is either rv_2, either subsumed by rv_2.
We conclude using transitivity of subsumption.

hence *A* : *subpath* (*red prb'*) (*subsumer sub*) *res* (*src re*) (*subs prb'*)
and *B* : *subpath* (*red prb'*) (*src re*) [*re*] (*tgt re*) (*subs prb'*)
using *subs-sub-rel-of[OF RB']* **by** (*auto simp add* : *sp-append-one sp-one*)

obtain c'
where *C* : *SE-star* (*confs prb'* rv_1) (*trace* (*ui-es res*) (*labeling* (*black prb'*)))
c'
and *D* : *SE* c' (*labeling* (*black prb'*) (*ui-edge re*)) *c*
using *2* **by** (*simp add* : *SE-star-append-one*) *blast*

obtain c''
where *E* : *SE* (*confs prb'* (*src re*)) (*labeling* (*black prb'*) (*ui-edge re*)) c''
using *subsum-step(6−8)*
‹*subpath* (*red prb'*) (*src re*) [*re*] (*tgt re*) (*subs prb'*)›
RB' finite-RedBlack.ex-se-succ[of prb' src re]
unfolding *finite-RedBlack-def*
by (*simp add* : *SE-star-one fst-of-sp-is-vert*) *blast*

have $c \sqsubseteq c''$
proof −
have $c' \sqsubseteq$ *confs prb'* (*src re*) **using** *2(1) A B C D* **by** *fast*

```
    thus ?thesis using D E SE-mono-for-sub by fast
  qed

  moreover
  have c'' ⊑ confs prb' (tgt re)
  proof -
    have subpath (red prb) (src re) [re] (tgt re) (subs prb)
    proof -
      have src re ∈ red-vertices prb'
      and  tgt re ∈ red-vertices prb'
      and  re ∈ edges (red prb')
      using B by (auto simp add : vertices-def sp-one)

      hence src re ∈ red-vertices prb
      and   tgt re ∈ red-vertices prb
      and   re ∈ edges (red prb)
      using subsum-step(3) by auto

      thus ?thesis
      using subs-sub-rel-of[OF subsum-step(1)]
      by (simp add : sp-one)
    qed

    thus ?thesis
    using subsum-step(2,3,6−8) E
    by (simp add : SE-star-one)
  qed

  moreover
  have confs prb' (tgt re) ⊑ confs prb' rv2
  proof -
    have tgt re = rv2 ∨ (tgt re,rv2) ∈ subs prb'
    using 2(2) rb-sp-append-one[OF RB'] by auto

    thus ?thesis
    proof (elim disjE)
      assume tgt re = rv2
      thus ?thesis
      by (simp add : subsums-refl)
    next
      assume (tgt re, rv2) ∈ (subs prb')
      thus ?thesis
      using sub-subsumed RB' by fastforce
    qed
  qed

  ultimately
  show ?case using subsums-trans subsums-trans by fast
qed
```

qed

next

— If *res* does not start at the newly subsumed red vertex, then either *res* is a subpath in the old red part, either it can be split into two parts res_1 and res_2 such that :

- res_1 is a subpath in the old red part from rv_1 to the newly subsumed vertex,
- res_2 is a subpath in the new red part from the newly subsumed vertex to rv_2.

assume $rv_1 \neq subsumee\ sub$

hence $subpath\ (red\ prb)\ rv_1\ res\ rv_2\ (subs\ prb) \vee$
$(\exists\ res_1\ res_2.\ res = res_1\ @\ res_2$
$\wedge\ res_1 \neq []$
$\wedge\ subpath\ (red\ prb)\ rv_1\ res_1\ (subsumee\ sub)\ (subs\ prb)$
$\wedge\ subpath\ (red\ prb')\ (subsumee\ sub)\ res_2\ rv_2\ (subs\ prb'))$
using *subsum-step(1,3,4)*
sp-in-extends-imp2
[*of subsumee sub subsumer sub red prb subs prb*]
by *auto*

thus *?thesis*
proof (*elim disjE exE conjE*)
— In the first case, we conclude using external IH.
assume $subpath\ (red\ prb)\ rv_1\ res\ rv_2\ (subs\ prb)$
thus *?thesis* **using** *subsum-step* **by** *simp*

next
— We call :

- c_1 the configuration obtained from the configuration at rv_1 by symbolic execution of res_1 and such that c is obtained from c_1 by symbolic execution of res_2,
- c_2 the configuration obtained from the configuration at the newly subsumed red vertex by symbolic execution of res_2.

We show that c is subsumed by c_2 and that c_2 is subsumed by the configuration at rv_2. We conclude by transitivity of subsumption.

fix $res_1\ res_2$

def $t\text{-}res_1 \equiv trace\ (ui\text{-}es\ res_1)\ (labeling\ (black\ prb'))$
def $t\text{-}res_2 \equiv trace\ (ui\text{-}es\ res_2)\ (labeling\ (black\ prb'))$

assume $res = res_1\ @\ res_2$
and $res_1 \neq []$

and *subpath* (*red prb*) rv_1 res_1 (*subsumee sub*) (*subs prb*)
and *subpath* (*red prb'*) (*subsumee sub*) res_2 rv_2 (*subs prb'*)

moreover
then obtain c_1 c_2
where *SE-star* (*confs prb'* rv_1) *t-res*$_1$ c_1
and *SE-star* c_1 *t-res*$_2$ *c*
and *SE-star* (*confs prb'* (*subsumee sub*)) *t-res*$_2$ c_2
using *subsum-step*(*1*,*3*,*5*,*6*−*8*) *RB'*
finite-RedBlack.*ex-SE-star-succ*[*of prb* rv_1 *t-res*$_1$]
finite-RedBlack.*ex-SE-star-succ*[*of prb' subsumee sub t-res*$_2$]
unfolding *finite-RedBlack-def t-res*$_1$*-def t-res*$_2$*-def*
by (*simp add* : *fst-of-sp-is-vert SE-star-append*) *blast*

ultimately
have $c \sqsubseteq c_2$
proof −
have $c_1 \sqsubseteq$ *confs prb'* (*subsumee sub*)
using *subsum-step*(*2*,*3*,*6*−*8*)
‹*subpath* (*red prb*) rv_1 res_1 (*subsumee sub*) (*subs prb*)›
‹*SE-star* (*confs prb'* rv_1) *t-res*$_1$ c_1›
by (*auto simp add* : *t-res*$_1$*-def t-res*$_2$*-def*)

thus *?thesis*
using ‹*SE-star* c_1 *t-res*$_2$ *c*›
‹*SE-star* (*confs prb'* (*subsumee sub*)) *t-res*$_2$ c_2›
SE-star-mono-for-sub
by *fast*
qed

moreover
— Here we have to proceed by backward induction on res_2.
have $c_2 \sqsubseteq$ *confs prb'* rv_2
using ‹*subpath* (*red prb'*) (*subsumee sub*) res_2 rv_2 (*subs prb'*)›
‹*SE-star* (*confs prb'* (*subsumee sub*)) *t-res*$_2$ c_2›
unfolding *t-res*$_2$*-def*
proof (*induct* res_2 *arbitrary* : rv_2 c_2 *rule* : *rev-induct*, *goal-cases*)
— If the suffix is empty, then either *subsumee sub* = rv_2, or (*subsumee sub*, rv_2) is in the new subsumption relation.
case (*1* rv_2 c_2)

hence *subsumee sub* = rv_2 ∨ (*subsumee sub*, rv_2)∈*subs prb'*
using *rb-Nil-sp*[*OF RB'*] **by** *simp*

thus *?case*
proof (*elim disjE*)
— In the first case, we have: *c* = *confs prb'* (*subsumee sub*) and *c* = *confs prb'* rv_2. We conclude by reflexivity of the subsumption.
assume *subsumee sub* = rv_2

thus *?thesis* **using** *1(2)* **by** (*simp add* : *subsums-refl*)
next
— In the second case, we have that c = *confs prb'* (*subsumee sub*) and *confs prb'* (*subsumee sub*) ⊑ *confs prb'* rv_2, qed.
assume (*subsumee sub*, rv_2) ∈ *subs prb'*
thus *?thesis* **using** *1(2)* *sub-subsumed*[*OF RB'*, *of* (*subsumee sub*, rv_2)]
by *simp*
qed

next
— Inductive case : the suffix has the form res_2 @ [*re*].
case (*2 re* res_2 rv_2 c_2)

— We call :

- c_3 the configuration obtained from the configuration at the newly subsumed red vertex. c_2 is obtained from c_3 by symbolic execution of *re*,
- c_4 the configuration obtained from the configuration at the source of *re* by symbolic execution of *re*.

By internal IH, we first show that c_3 is subsumed by the configuration at the source of *re*. Thus c_2 is subsumed by c_4, by monotonicity of symbolic execution w.r.t. subsumption.
Then, we show that c_4 is subsumed by the configuration at the target of *re*, using the external IH.
Finally, we show that the configuration at the target of *re* is subsumed by the configuration at rv_2, by observing that either *tgt re* = rv_2, or that (*tgt re*, rv_2) is in the new subsumption relation.
We conclude by transitivity of the subsumption relation.

have *A* : *subpath* (*red prb'*) (*subsumee sub*) res_2 (*src re*) (*subs prb'*)
and *B* : *subpath* (*red prb'*) (*src re*) [*re*] rv_2 (*subs prb'*)
using *2(2)* *subs-wf-sub-rel*[*OF RB'*] *subs-wf-sub-rel-of*[*OF RB'*]
by (*simp-all only*: *wf-sub-rel.sp-append-one*)
(*simp add* : *wf-sub-rel.sp-one wf-sub-rel-of-def*)

obtain c_3
where *C* : *SE-star* (*confs prb'* (*subsumee sub*)) (*trace* (*ui-es* res_2) (*labeling* (*black prb'*))) c_3
and *D* : *SE* c_3 (*labeling* (*black prb'*) (*ui-edge re*)) c_2
using *2(3)* *subsum-step(6−8)* *RB'*
finite-RedBlack.ex-se-succ[*of prb' src re*]
by (*simp add* : *SE-star-append-one*) *blast*

obtain c_4
where *E* : *SE* (*confs prb'* (*src re*)) (*labeling* (*black prb'*) (*ui-edge re*)) c_4
using *subsum-step(6−8)* *RB' B*
finite-RedBlack.ex-se-succ[*of prb' src re*]
unfolding *finite-RedBlack-def*
by (*simp add* : *fst-of-sp-is-vert SE-star-append*) *blast*

have $c_2 \sqsubseteq c_4$
proof −
  have $c_3 \sqsubseteq$ *confs prb′* (*src re*) **using** *2*(*1*) *A C* **by** *fast*

  thus *?thesis* **using** *D E SE-mono-for-sub* **by** *fast*
qed

moreover
have $c_4 \sqsubseteq$ *confs prb′* (*tgt re*)
proof −
  have *subpath* (*red prb*) (*src re*) [*re*] (*tgt re*) (*subs prb*)
  proof −
    have *src re* ∈ *red-vertices prb′*
    and *tgt re* ∈ *red-vertices prb′*
    and *re* ∈ *edges* (*red prb′*)
    using *B* **by** (*auto simp add* : *vertices-def sp-one*)

    hence *src re* ∈ *red-vertices prb*
    and *tgt re* ∈ *red-vertices prb*
    and *re* ∈ *edges* (*red prb*)
    using *subsum-step*(*3*) **by** *auto*

    thus *?thesis*
    using *subs-sub-rel-of*[*OF subsum-step*(*1*)]
    by (*simp add* : *sp-one*)
  qed

  thus *?thesis*
  using *subsum-step*(*2*,*3*,*6*−*8*) *E*
  by (*simp add* : *SE-star-one*)
qed

moreover
have *confs prb′* (*tgt re*) ⊑ *confs prb′* rv_2
proof −
  have *tgt re* = rv_2 ∨ (*tgt re*, rv_2) ∈ (*subs prb′*)
  using *subsum-step 2 rb-sp-append-one*[*OF RB′*, *of subsumee sub* res_2 *re*]
  by (*auto simp add* : *vertices-def subpath-def*)

  thus *?thesis*
  proof (*elim disjE*)
    assume *tgt re* = rv_2
    thus *?thesis* **by** (*simp add* : *subsums-refl*)
  next
    assume (*tgt re*, rv_2) ∈ (*subs prb′*)
    thus *?thesis*
    using *sub-subsumed RB′*
    by *fastforce*

```
        qed
      qed

      ultimately
      show ?case using subsums-trans subsums-trans by fast
    qed

    ultimately
    show ?thesis by (rule subsums-trans)
  qed
qed

next
  case (abstract-step prb rv c_a prb' rv_1 res rv_2 c)

  show ?case
  proof (case-tac rv_1 = rv, goal-cases)
```

— We first suppose that rv_1 is the red vertex where the abstraction took place. In this case, we have that *res* is empty and $rv_2 = rv_1$. Hence c is the configuration at rv_2 (after abstraction). We conclude using reflexivity of subsumption.

```
    case 1

    moreover
    hence res = []
    using abstract-step
          sp-from-de-empty[of rv_1 subs prb red prb res rv_2]
    by simp

    moreover
    have rv_2 = rv
    proof −
      have rv_1 = rv_2 ∨ (rv_1, rv_2) ∈ (subs prb)
      using abstract-step ‹res = []›
            rb-Nil-sp[OF RedBlack.abstract-step[OF abstract-step(1,3)]]
      by simp

      moreover
      have (rv_1, rv_2) ∉ (subs prb)
      using abstract-step 1
      unfolding Ball-def subsumees-conv
      by (intro notI) blast

      ultimately
      show ?thesis using 1 by simp
    qed

    ultimately
    show ?thesis using abstract-step(5) by (simp add : subsums-refl)
  next
```

— Suppose that rv_1 is not the red vertex where the subsumption took place.
case *2*

show *?thesis*
proof (*case-tac* $rv_2 = rv$)
— We first suppose that rv_2 is the newly abstracted red vertex. Then we have that the new configuration at rv_2 subsums the old configuration at this red vertex. We conclude by use of IH and transitivity of subsumption.
assume $rv_2 = rv$

hence $confs\ prb\ rv_2 \sqsubseteq confs\ prb'\ rv_2$
using *abstract-step* **by** (*simp add* : *abstract-def*)

moreover
have $c \sqsubseteq confs\ prb\ rv_2$ **using** *abstract-step 2* **by** *auto*

ultimately
show *?thesis* **using** *subsums-trans* **by** *fast*
next
assume $rv_2 \neq rv$ **thus** *?thesis* **using** *abstract-step 2* **by** *simp*
qed
qed

next
— Strengthening a red vertex does not affect the red part, thus this case is trivial.

case *strengthen-step* **thus** *?case* **by** *simp*
qed

A.12.6 Properties about marking.

A configuration which is indeed satisfiable can not be marked.

lemma *sat-not-marked* :
assumes *RedBlack prb*
assumes $rv \in red\text{-}vertices\ prb$
assumes *sat* (*confs prb rv*)
shows $\neg\ marked\ prb\ rv$
using *assms*
proof (*induct prb arbitrary* : *rv*)
case *base* **thus** *?case* **by** *simp*
next
case (*se-step prb re c prb'*)

hence $rv \in red\text{-}vertices\ prb \lor rv = tgt\ re$ **by** (*auto simp add* : *vertices-def*)

thus *?case*
proof (*elim disjE*, *goal-cases*)
case *1*
moreover

```
    hence rv ≠ tgt re using se-step(3) by (auto simp add : vertices-def)
    ultimately
    show ?thesis using se-step by (elim conjE) auto
  next
    case 2

    moreover
    hence sat (confs prb (src re)) using se-step(3,5) SE-sat-imp-sat by auto

    ultimately
    show ?thesis using se-step(2,3) by (elim conjE) auto
  qed
next
  case (mark-step prb rv' prb')

  moreover
  hence rv ≠ rv' and (rv,rv') ∉ subs prb
  using sub-subsumed[OF mark-step(1), of (rv,rv')] unsat-subs-unsat by auto

  ultimately
  show ?case by auto
next
  case subsum-step thus ?case by auto

next
  case (abstract-step prb rv' c_a prb') thus ?case by (case-tac rv' = rv) simp+

next
  case strengthen-step thus ?case by simp
qed
```

On the other hand, a red-location which is marked unsat is indeed logically unsatisfiable.

```
lemma
  assumes RedBlack prb
  assumes rv ∈ red-vertices prb
  assumes marked prb rv
  shows   ¬ sat (confs prb rv)
using assms
proof (induct prb arbitrary : rv)
  case base thus ?case by simp
next
  case (se-step prb re c prb')

  hence rv ∈ red-vertices prb ∨ rv = tgt re by (auto simp add : vertices-def)

  thus ?case
  proof (elim disjE, goal-cases)
    case 1
```

moreover
hence *rv* ≠ *tgt re* **using** *se-step*(*3*) **by** *auto*
hence *marked prb rv* **using** *se-step* **by** *auto*

ultimately
have ¬ *sat* (*confs prb rv*) **by** (*rule se-step*(*2*))

thus *?thesis* **using** *se-step*(*3*) ‹*rv* ≠ *tgt re*› **by** *auto*
next
case *2*

moreover
hence *marked prb* (*src re*) **using** *se-step*(*3*,*5*) **by** *auto*

ultimately
have ¬ *sat* (*confs prb* (*src re*)) **using** *se-step*(*2*,*3*) **by** *auto*

thus *?thesis* **using** *se-step*(*3*) ‹*rv* = *tgt re*› *unsat-imp-SE-unsat* **by** (*elim conjE*) *auto*
qed
next
case (*mark-step prb rv′ prb′*) **thus** *?case* **by** (*case-tac rv′* = *rv*) *auto*
next
case *subsum-step* **thus** *?case* **by** *simp*

next
case (*abstract-step* - *rv′* -) **thus** *?case* **by** (*case-tac rv′* = *rv*) *simp+*

next
case *strengthen-step* **thus** *?case* **by** *simp*
qed

Red vertices involved in subsumptions are not marked.

lemma *subsumee-not-marked* :
assumes *RedBlack prb*
assumes *sub* ∈ *subs prb*
shows ¬ *marked prb* (*subsumee sub*)
using *assms*
proof (*induct prb*)
case *base* **thus** *?case* **by** *simp*
next
case (*se-step prb re c prb′*)

moreover
hence *subsumee sub* ≠ *tgt re*
using *subs-wf-sub-rel-of*[*OF se-step*(*1*)]
by (*elim conjE*, *auto simp add* : *wf-sub-rel-of-def sub-rel-of-def*)

```
  ultimately
  show ?case by auto
next
  case mark-step thus ?case by auto
next
  case subsum-step thus ?case by auto

next
  case abstract-step thus ?case by auto

next
  case strengthen-step thus ?case by simp
qed


lemma subsumer-not-marked :
  assumes RedBlack prb
  assumes sub ∈ subs prb
  shows   ¬ marked prb (subsumer sub)
using assms
proof (induct prb)
  case base thus ?case by simp
next
  case (se-step prb re c prb')

  moreover
  hence subsumer sub ≠ tgt re
  using subs-wf-sub-rel-of[OF se-step(1)]
  by (elim conjE, auto simp add : wf-sub-rel-of-def sub-rel-of-def)

  ultimately
  show ?case by auto
next
  case (mark-step prb rv prb') thus ?case by auto
next
  case (subsum-step prb sub' prb') thus ?case by auto

next
  case abstract-step thus ?case by simp

next
  case strengthen-step thus ?case by simp
qed
```

If the target of a red edge is not marked, then its source is also not marked.

```
lemma tgt-not-marked-imp :
  assumes RedBlack prb
  assumes re ∈ edges (red prb)
```

```
  assumes ¬ marked prb (tgt re)
  shows   ¬ marked prb (src re)
using assms
proof (induct prb arbitrary : re)
  case base thus ?case by simp
next
  case se-step thus ?case by (force simp add : vertices-def image-def)
next
  case (mark-step prb rv prb' re) thus ?case by (case-tac tgt re = rv) auto
next
  case subsum-step thus ?case by simp

next
  case abstract-step thus ?case by simp

next
  case strengthen-step thus ?case by simp
qed
```

Given a red sub-path leading from red location *rv1* to red location *rv2*, if *rv2* is not marked, then *rv1* is also not marked (this lemma is not used).

```
lemma
  assumes RedBlack prb
  assumes subpath (red prb) rv1 res rv2 (subs prb)
  assumes ¬ marked prb rv2
  shows   ¬ marked prb rv1
using assms
proof (induct res arbitrary : rv1)
  case Nil

  hence rv1 = rv2 ∨ (rv1,rv2) ∈ subs prb by (simp add : rb-Nil-sp)

  thus ?case
  proof (elim disjE, goal-cases)
    case 1 thus ?case using Nil by simp
  next
    case 2 show ?case using Nil subsumee-not-marked[OF Nil(1) 2] by simp
  qed
next
  case (Cons re res)

  thus ?case
  unfolding rb-sp-Cons[OF Cons(2), of rv1 re res rv2]
  proof (elim conjE disjE, goal-cases)
    case 1

    moreover
    hence ¬ marked prb (tgt re) by simp
```

moreover
have *re* $\in$ *edges* (*red prb*) **using** *Cons*(*3*) *rb-sp-Cons*[*OF Cons*(*2*), *of* rv_1 *re res* rv_2] **by** *fast*

ultimately
show *?thesis* **using** *tgt-not-marked-imp*[*OF Cons*(*2*)] **by** *fast*
next
case *2* **thus** *?thesis* **using** *subsumee-not-marked*[*OF Cons*(*2*)] **by** *fastforce*
qed
qed

A.12.7 Fringe of a red-black graph

We have stated and proved a number of properties of red-black graphs. In the end, we are mainly interested in proving that the set of paths of such red-black graphs are subsets of the set of feasible paths of their black part. Before defining the set of paths of red-black graphs, we first introduce the intermediate concept of *fringe* of the red part. Intuitively, the fringe is the set of red vertices from which we can approximate more precisely the set of feasible paths of the black part. This includes red vertices that have not been subsumed yet, that are not marked and from which some black edges have not been yet symbolically executed (i.e. that have no red counterpart from these red vertices).

Definition

The fringe is the set of red locations from which there exist black edges that have not been followed yet.

definition *fringe* ::
('*vert*, '*var*, '*d*, '*x*) *pre-RedBlack-scheme* $\Rightarrow$ ('*vert* $\times$ *nat*) *set*
where
fringe prb $\equiv$ {*rv* $\in$ *red-vertices prb*.
rv $\notin$ *subsumees* (*subs prb*) $\wedge$
$\neg$ *marked prb rv* $\wedge$
ui-edge ` *out-edges* (*red prb*) *rv* $\subset$ *out-edges* (*black prb*) (*fst rv*)}

Fringe of an empty red-part

At the beginning of the analysis, i.e. when the red part is empty, the fringe consists of the red root.

lemma *fringe-of-empty-red* :
assumes *edges* (*red prb*) = {}
assumes *subs prb* = {}
assumes *marked prb* = (λ *rv*. *False*)
assumes *out-edges* (*black prb*) (*fst* (*root* (*red prb*))) $\neq$ {}
shows *fringe prb* = {*root* (*red prb*)}
using *assms* **by** (*auto simp add* : *fringe-def vertices-def*)

Evolution of the fringe after extension

Simplification lemmas for the fringe of the new red-black graph after adding an edge by symbolic execution. If the configuration from which symbolic execution is performed is not marked yet, and if there exists black edges going out of the target of the executed edge, the target of the new red edge enters the fringe. Moreover, if there still exist black edges that have no red counterpart yet at the source of the new edge, then its source was and stays in the fringe.

lemma *seE-fringe1* :
  assumes *sub-rel-of* (*red prb*) (*subs prb*)
  assumes *se-extends prb re c′ prb′*
  assumes ¬ *marked prb* (*src re*)
  assumes *ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*))
  assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) ≠ {}
  shows *fringe prb′* = *fringe prb* ∪ {*tgt re*}
unfolding *set-eq-iff Un-iff singleton-iff*
proof (*intro allI iffI*, *goal-cases*)
  case (*1 rv*)

  moreover
  hence *rv* ∈ *red-vertices prb* ∨ *rv* = *tgt re*
  using *assms*(*2*) **by** (*auto simp add* : *fringe-def vertices-def*)

  ultimately
  show *?case* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def*)
next
  case (*2 rv*)

  hence *rv* ∈ *red-vertices prb′* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def vertices-def*)

  moreover
  have *rv* ∉ *subsumees* (*subs prb′*)
  using *2*
  proof (*elim disjE*)
    assume *rv* ∈ *fringe prb* **thus** *?thesis* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def*)
  next
    assume *rv* = *tgt re* **thus** *?thesis*
    using *assms*(*1*,*2*) **unfolding** *sub-rel-of-def* **by** *force*
  qed

  moreover
  have *ui-edge* ‘ (*out-edges* (*red prb′*) *rv*) ⊂ *out-edges* (*black prb′*) (*fst rv*)
  using *2*
  proof (*elim disjE*)

```
    assume rv ∈ fringe prb

    thus ?thesis
    proof (case-tac rv = src re)
      assume rv = src re thus ?thesis using assms(2,4) by auto
    next
      assume rv ≠ src re thus ?thesis
      using assms(2) ‹rv ∈ fringe prb›
      by (auto simp add : fringe-def)
    qed
  next
    assume rv = tgt re thus ?thesis
    using assms(2,5) extends-tgt-out-edges[of re red prb red prb'] by (elim conjE)
auto
  qed

  moreover
  have ¬ marked prb' rv
  using 2
  proof (elim disjE, goal-cases)
    case 1

    moreover
    hence rv ≠ tgt re using assms(2) by (auto simp add : fringe-def)

    ultimately
    show ?thesis using assms(2) by (auto simp add : fringe-def)
  next
    case 2 thus ?thesis using assms(2,3) by auto
  qed

  ultimately
  show ?case by (simp add : fringe-def)
qed
```

If the source of the new edge is marked, then its target does not enter the fringe (and the source was not part of it in the first place).

```
lemma seE-fringe2 :
  assumes se-extends prb re c prb'
  assumes marked prb (src re)
  shows   fringe prb' = fringe prb
unfolding set-eq-iff Un-iff singleton-iff
proof (intro allI iffI, goal-cases)
  case (1 rv)

  thus ?case
  unfolding fringe-def mem-Collect-eq
  using assms
  proof (intro conjI, goal-cases)
```

```
    case 1 thus ?case by (auto simp add : fringe-def vertices-def)
  next
    case 2 thus ?case by auto
  next
    case 3

    moreover
    hence rv ≠ tgt re by auto

    ultimately
    show ?case by auto
  next
    case 4 thus ?case by auto
  qed
next
  case (2 rv)

  thus ?case unfolding fringe-def mem-Collect-eq
  using assms
  proof (intro conjI, goal-cases)
    case 1 thus ?case by (auto simp add : vertices-def)
  next
    case 2 thus ?case by auto
  next
    case 3
    moreover
    hence rv ≠ tgt re by auto
    ultimately
    show ?case by auto
  next
    case 4 thus ?case by auto
  qed
qed
```

If there exists no black edges going out of the target of the new edge, then this target does not enter the fringe.

```
lemma seE-fringe3 :
  assumes se-extends prb re c' prb'
  assumes ui-edge ` (out-edges (red prb') (src re)) ⊂ out-edges (black prb) (fst (src
re))
  assumes out-edges (black prb) (fst (tgt re)) = {}
  shows   fringe prb' = fringe prb
unfolding set-eq-iff Un-iff singleton-iff
proof (intro allI iffI, goal-cases)
  case (1 rv)

  thus ?case using assms(1,3)
  unfolding fringe-def mem-Collect-eq
  proof (intro conjI, goal-cases)
```

case *1* **thus** *?case* **by** (*auto simp add* : *fringe-def vertices-def*)
next
case *2* **thus** *?case* **by** (*auto simp add* : *fringe-def*)
next
case *3* **thus** *?case* **by** (*case-tac rv* = *tgt re*) (*auto simp add* : *fringe-def*)
next
case *4* **thus** *?case* **by** (*auto simp add* : *fringe-def*)
qed

next
case (*2 rv*)

moreover
hence *rv* ∈ *red-vertices prb′*
and *rv* ≠ *tgt re*
using *assms*(*1*) **by** (*auto simp add* : *fringe-def vertices-def*)

moreover
have *ui-edge* ‘ (*out-edges* (*red prb′*) *rv*) ⊂ *out-edges* (*black prb*) (*fst rv*)
proof (*case-tac rv* = *src re*)
assume *rv* = *src re* **thus** *?thesis* **using** *assms*(*2*) **by** *simp*
next
assume *rv* ≠ *src re*
thus *?thesis* **using** *assms*(*1*) *2*
by (*auto simp add* : *fringe-def*)
qed

ultimately
show *?case* **using** *assms*(*1*) **by** (*auto simp add* : *fringe-def*)
qed

If all possible black edges have been executed from the source of the new edge after the extension, then the source is removed from the fringe.

lemma *seE-fringe4* :
assumes *sub-rel-of* (*red prb*) (*subs prb*)
assumes *se-extends prb re c′ prb′*
assumes ¬ *marked prb* (*src re*)
assumes ¬ (*ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*)))
assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) ≠ {}
shows *fringe prb′* = *fringe prb* − {*src re*} ∪ {*tgt re*}
unfolding *set-eq-iff Un-iff singleton-iff Diff-iff*
proof (*intro allI iffI*, *goal-cases*)
case (*1 rv*)

moreover
hence *rv* ∈ *red-vertices prb* ∨ *rv* = *tgt re*
and *rv* ≠ *src re*
using *assms*(*2*,*3*,*4*,*5*) **by** (*auto simp add* : *fringe-def vertices-def*)

ultimately
show *?case* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def*)

next
case (*2 rv*)

hence $rv \in$ *red-vertices prb′* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def vertices-def*)

moreover
have $rv \notin$ *subsumees* (*subs prb′*)
using *2*
proof (*elim disjE*)
assume $rv \in$ *fringe prb* $\wedge$ $rv \neq$ *src re*
thus *?thesis* **using** *assms*(*2*) **by** (*auto simp add* : *fringe-def*)
next
assume $rv =$ *tgt re* **thus** *?thesis*
using *assms*(*1*,*2*) **unfolding** *sub-rel-of-def* **by** *fastforce*
qed

moreover
have *ui-edge* ‘ (*out-edges* (*red prb′*) *rv*) $\subset$ *out-edges* (*black prb′*) (*fst rv*)
using *2*
proof (*elim disjE*)
assume $rv \in$ *fringe prb* $\wedge$ $rv \neq$ *src re* **thus** *?thesis*
using *assms*(*2*) **by** (*auto simp add* : *fringe-def*)
next
assume $rv =$ *tgt re* **thus** *?thesis*
using *assms*(*2*,*5*) *extends-tgt-out-edges*[*of re red prb red prb′*] **by** (*elim conjE*) *auto*
qed

moreover
have ¬ *marked prb′ rv*
using *2*
proof (*elim disjE*, *goal-cases*)
case *1*

moreover
hence $rv \neq$ *tgt re* **using** *assms* **by** (*auto simp add* : *fringe-def*)

ultimately
show *?thesis*
using *assms 1* **by** (*auto simp add* : *fringe-def*)
next
case *2* **thus** *?thesis* **using** *assms* **by** *auto*
qed

ultimately
show *?case* **by** (*simp add* : *fringe-def*)
qed

If all possible black edges have been executed from the source of the new edge after the extension, then this source is removed from the fringe.

lemma *seE-fringe5* :
assumes *se-extends prb re c′ prb′*
assumes ¬ (*ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*)))
assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) = {}
shows *fringe prb′* = *fringe prb* − {*src re*}
unfolding *set-eq-iff Un-iff singleton-iff Diff-iff*
proof (*intro allI iffI*, *goal-cases*)
case (*1 rv*)

moreover
hence *rv* ∈ *red-vertices prb* **and** *rv* ≠ *src re*
using *assms* **by** (*auto simp add* : *fringe-def vertices-def*)

moreover
hence ¬ *marked prb rv*
proof (*intro notI*)
assume *marked prb rv*

have *marked prb′ rv*
proof −
have *rv* ≠ *tgt re* **using** *assms*(*1*) ‹*rv* ∈ *red-vertices prb*› **by** *auto*

thus *?thesis* **using** *assms*(*1*) ‹*marked prb rv*› **by** *auto*
qed

thus *False* **using** *1* **by** (*auto simp add* : *fringe-def*)
qed

ultimately
show *?case* **using** *assms*(*1*) **by** (*auto simp add* : *fringe-def*)

next
case (*2 rv*)

hence *rv* ∈ *red-vertices prb′* **using** *assms*(*1*) **by** (*auto simp add* : *fringe-def vertices-def*)

moreover
have *rv* ∉ *subsumees* (*subs prb′*) **using** *2 assms*(*1*) **by** (*auto simp add* : *fringe-def*)

moreover
have *ui-edge* ‘ (*out-edges* (*red prb′*) *rv*) ⊂ *out-edges* (*black prb′*) (*fst rv*)

using *2 assms*(*1*) **by** (*auto simp add* : *fringe-def*)

moreover
have ¬ *marked prb′ rv*
proof −
have $rv \neq tgt\ re$ **using** *assms*(*1*) *2* **by** (*auto simp add* : *fringe-def*)

thus *?thesis* **using** *assms*(*1*) *2* **by** (*auto simp add* : *fringe-def*)
qed

ultimately
show *?case* **by** (*simp add* : *fringe-def*)
qed

Adding a subsumption to the subsumption relation removes the first member of the subsumption from the fringe.

lemma *subsumE-fringe* :
assumes *subsum-extends prb sub prb′*
shows *fringe prb′* = *fringe prb* − {*subsumee sub*}
using *assms* **by** (*auto simp add* : *fringe-def*)

A.12.8 Red-black sub-paths and paths

The set of red-black sub-paths starting in red location *rv* is the union of :

- the set of black sub-paths that have a red counterpart starting at *rv* and leading to a non-marked red location,
- the set of black sub-paths that have a prefix represented in the red part starting at *rv* and leading to an element of the fringe. Moreover, the remainings of these black sub-paths must have no non-empty counterpart in the red part. Otherwise, the set of red-black paths would simply be the set of paths of the black part.

definition *RedBlack-subpaths-from* ::
(*′vert*, *′var*, *′d*, *′x*) *pre-RedBlack-scheme* ⇒ (*′vert* × *nat*) ⇒ *′vert edge list set*
where
RedBlack-subpaths-from prb rv ≡
ui-es ‘ {*res*. ∃ *rv′*. *subpath* (*red prb*) *rv res rv′* (*subs prb*) ∧ ¬ *marked prb rv′*}
∪ {$ui\text{-}es\ res_1$ @ bes_2
| $res_1\ bes_2$. ∃ rv_1. $rv_1 \in fringe\ prb$
∧ *subpath* (*red prb*) *rv* $res_1\ rv_1$ (*subs prb*)
∧ ¬ (∃ $res_{21}\ bes_{22}$. $bes_2 = ui\text{-}es\ res_{21}$ @ bes_{22}
∧ $res_{21} \neq []$
∧ *subpath-from* (*red prb*) $rv_1\ res_{21}$ (*subs prb*))
∧ *Graph.subpath-from* (*black prb*) (*fst* rv_1) bes_2}

Red-black paths are red-black sub-paths starting at the root of the red part.

```
abbreviation RedBlack-paths ::
  ('vert, 'var, 'd, 'x) pre-RedBlack-scheme ⇒ 'vert edge list set
where
  RedBlack-paths prb ≡ RedBlack-subpaths-from prb (root (red prb))
```

When the red part is empty, the set of red-black sub-paths starting at the red root is the set of black paths.

```
lemma (in finite-RedBlack) base-RedBlack-paths :
  assumes fst (root (red prb)) = init (black prb)
  assumes edges (red prb) = {}
  assumes subs prb = {}
  assumes confs prb (root (red prb)) = init-conf prb
  assumes marked prb = (λ rv. False)
  assumes strengthenings prb = (λ rv. (λ σ. True))

  shows   RedBlack-paths prb = Graph.paths (black prb)

proof −

  show ?thesis
  unfolding set-eq-iff
  proof (intro allI iffI)

    fix bes

    assume bes ∈ RedBlack-subpaths-from prb (root (red prb))

    thus bes ∈ Graph.paths (black prb)
    unfolding  RedBlack-subpaths-from-def Un-iff
    proof (elim disjE exE conjE, goal-cases)
      case 1

      hence bes = [] using assms by (auto simp add: subpath-def)

      thus ?thesis
      by (auto simp add : Graph.subpath-def vertices-def)
    next
      case 2

      then obtain res1 bes2 rv where bes = ui-es res1 @ bes2
                             and   rv ∈ fringe prb
                           and   subpath (red prb) (root (red prb)) res1 rv (subs prb)
                             and   Graph.subpath-from (black prb) (fst rv) bes2
      by blast

      moreover
      hence res1 = [] using assms by (simp add : subpath-def)

      ultimately
```

show *?thesis* **using** *assms* ‹*rv* ∈ *fringe prb*› **by** (*simp add* : *fringe-def vertices-def*)
qed
next
fix *bes*

assume *bes* ∈ *Graph.paths* (*black prb*)

show *bes* ∈ *RedBlack-subpaths-from prb* (*root* (*red prb*))
proof (*case-tac out-edges* (*black prb*) (*init* (*black prb*)) = {})
assume *out-edges* (*black prb*) (*init* (*black prb*)) = {}

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x*=[] **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac ?x*=*root* (*red prb*) **in** *exI*)
proof (*intro conjI*)
show *subpath* (*red prb*) (*root* (*red prb*)) [] (*root* (*red prb*)) (*subs prb*)
using *assms*(*3*) **by** (*simp add* : *sub-rel-of-def subpath-def vertices-def*)
next
show ¬ *marked prb* (*root* (*red prb*)) **using** *assms*(*5*) **by** *simp*
next
show *bes* = *ui-es* []
using ‹*bes* ∈ *Graph.paths* (*black prb*)›
‹*out-edges* (*black prb*) (*init* (*black prb*)) = {}›
by (*cases bes*) (*auto simp add* : *Graph.sp-Cons*)
qed

next
assume *out-edges* (*black prb*) (*init* (*black prb*)) ≠ {}

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
proof (*intro disjI2*, *rule-tac ?x*=[] **in** *exI*, *rule-tac ?x*=*bes* **in** *exI*, *intro conjI*, *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2*

show *?case*
unfolding *Bex-def*
proof (*rule-tac ?x*=*root* (*red prb*) **in** *exI*, *intro conjI*, *goal-cases*)
show *root* (*red prb*) ∈ *fringe prb*
using *assms*(*1*−*3*,*5*) ‹*out-edges* (*black prb*) (*init* (*black prb*)) ≠ {}› *fringe-of-empty-red*
by *fastforce*
next

```
          show subpath (red prb) (root (red prb)) [] (root (red prb)) (subs prb)
          using subs-sub-rel-of[OF RedBlack.base[OF assms(1−6)]]
          by (simp add : subpath-def vertices-def sub-rel-of-def)
        next
          case 3
          show ?case
          proof (intro notI, elim exE conjE)
            fix res_21 bes_22 rv

            assume bes = ui-es res_21 @ bes_22
            and    res_21 ≠ []
            and    subpath (red prb) (root (red prb)) res_21 rv (subs prb)

            moreover
            hence res_21 = [] using assms by (simp add : subpath-def)

            ultimately
            show False by (elim notE)
          qed
        next
          case 4 show ?case using assms ‹bes ∈ Graph.paths (black prb)› by simp
        qed
      qed
    qed
  qed
qed
```

Red-black sub-paths and paths are sub-paths and paths of the black part.

```
lemma RedBlack-subpaths-are-black-subpaths :
  assumes RedBlack prb
  shows   RedBlack-subpaths-from prb rv ⊆ Graph.subpaths-from (black prb) (fst
rv)
unfolding subset-iff mem-Collect-eq RedBlack-subpaths-from-def Un-iff image-def
Bex-def
proof (intro allI impI, elim disjE exE conjE, goal-cases)
  case (1 bes res rv') thus ?case using assms red-sp-imp-black-sp by blast
next
  case (2 bes res_1 bes_2 rv_1 bv_2) thus ?case
  using red-sp-imp-black-sp[OF assms, of rv  res_1 rv_1]
  by (rule-tac ?x=bv_2 in exI) (auto simp add : Graph.sp-append)
qed

lemma RedBlack-paths-are-black-paths :
  assumes RedBlack prb
  shows   RedBlack-paths prb ⊆ Graph.paths (black prb)
using assms
      RedBlack-subpaths-are-black-subpaths[of prb root (red prb)]
      consistent-roots[of prb]
```

by *simp*

A.12.9 Preservation of feasible paths

The following theorem states that we do not loose feasible paths using our five operators, and moreover, configurations *c* at the end of feasible red paths in some graph *prb* will have corresponding feasible red paths in successors that lead to configurations that subsume *c*. As a corollary, our calculus is correct w.r.t. to execution.

theorem (**in** *finite-RedBlack*) *feasible-subpaths-preserved* :
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  shows *feasible-subpaths-from* (*black prb*) (*confs prb rv*) (*fst rv*)
    $\subseteq$ *RedBlack-subpaths-from prb rv*
using *assms finite-RedBlack*
proof (*induct prb arbitrary* : *rv*)

— Base case : the red part is empty. In this case, *rv* is the root of the red part. Hence, the set of feasible subpaths starting at *fst rv* is the set of feasible paths of the black part. We conclude using the fact that when the red part is empty, the set of red-black subpaths is the set of paths of the black part, which includes feasible paths.

  case (*base prb rv*)

  moreover
  hence *rv* = *root* (*red prb*) **by** (*simp add* : *vertices-def*)

  moreover
  hence *feasible-subpaths-from* (*black prb*) (*confs prb rv*) (*fst rv*)
    = *feasible-paths* (*black prb*) (*confs prb* (*root* (*red prb*)))
  using *base* **by** *simp*

  moreover
  have *out-edges* (*black prb*) (*fst* (*root* (*red prb*))) = {} $\vee$
    ui-edge ‘ *out-edges* (*red prb*) (*root* (*red prb*)) $\subset$ *out-edges* (*black prb*) (*fst* (*root* (*red prb*)))
  using *base* **by** *auto*

  ultimately
  show *?case*
  using *finite-RedBlack.base-RedBlack-paths*[*of prb*]
  by (*auto simp only* : *finite-RedBlack-def*)

next

— Adding an edge by symbolic execution.

case (*se-step prb re c prb′ rv*)

have *RB′* : *RedBlack prb′* **by** (*rule RedBlack.se-step*[*OF se-step*(*1*,*3*)])

show *?case*
unfolding *subset-iff*
proof (*intro allI impI*)

fix *bes*

assume *bes* ∈ *feasible-subpaths-from* (*black prb′*) (*confs prb′ rv*) (*fst rv*)

have *rv* ∈ *red-vertices prb* ∨ *rv* = *tgt re*
using *se-step*(*3*,*4*) **by** (*auto simp add* : *vertices-def*)

thus *bes* ∈ *RedBlack-subpaths-from prb′ rv*
proof (*elim disjE*)

— We first suppose that *bes* does not start at the target of the new edge. In this case, we can use the IH to show that *bes* is a red-black subpath in the old red-black graph. We then proceed by case distinction.

assume *rv* ∈ *red-vertices prb*

moreover
hence *rv* ≠ *tgt re* **using** *se-step* **by** *auto*

ultimately
have *bes* ∈ *RedBlack-subpaths-from prb rv*
using *se-step* ‹*bes* ∈ *feasible-subpaths-from* (*black prb′*) (*confs prb′ rv*) (*fst rv*)›
by *fastforce*

thus *?thesis*
apply (*subst* (*asm*) *RedBlack-subpaths-from-def*)
unfolding *Un-iff image-def Bex-def mem-Collect-eq*
proof (*elim disjE exE conjE*)

— Suppose that *bes* is entirely represented in the old red part. Then it is also entirely represented in the new red part, qed.

fix *res rv′*

assume *bes* = *ui-es res*
and *subpath* (*red prb*) *rv res rv′* (*subs prb*)
and ¬ *marked prb rv′*

moreover

hence ¬ *marked prb′ rv′*
using *se-step*(*3*) *lst-of-sp-is-vert*[*of red prb rv res rv′ subs prb*]
by (*elim conjE*) *auto*

ultimately
show *?thesis*
using *se-step*(*3*) *sp-in-extends-w-subs*[*of re red prb red prb′ rv res rv′ subs prb*]
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
by (*intro disjI1*, *rule-tac ?x=res* **in** *exI*, *intro conjI*)
(*rule-tac ?x=rv′* **in** *exI*, *auto*)

next

— Suppose that *bes* is not entirely represented in the old red part, i.e. *bes* is of the form *ui-es* res_1 @ bes_2 where res_1 is a (maximal) red subpath (leading to a non-marked element rv_1 of the old fringe) and bes_2 is black subpath (starting at the black vertex represented by rv_1. We then proceed by distinguishing the cases where the rv_1 is the source of the new edge or is an "old" red vertex.

fix res_1 bes_2 rv_1 *bl*

assume *A* : *bes* = *ui-es* res_1 @ bes_2
and *B* : rv_1 ∈ *fringe prb*
and *C* : *subpath* (*red prb*) *rv* res_1 rv_1 (*subs prb*)

and *E* : ¬ (∃ res_{21} bes_{22}. bes_2 = *ui-es* res_{21} @ bes_{22}
∧ $res_{21} \neq []$
∧ *subpath-from* (*red prb*) rv_1 res_{21} (*subs prb*))
and *F* : *Graph.subpath* (*black prb*) (*fst* rv_1) bes_2 *bl*

hence $rv_1 \neq$ *tgt re* **using** *se-step* **by** (*auto simp add* : *fringe-def*)

show *?thesis*
proof (*case-tac* rv_1 = *src re*)

— If rv_1 is the source of the new edge, we proceed by cases on the black suffix.

assume rv_1 = *src re*

show *?thesis*
proof (*case-tac* bes_2 = [])
— If the black suffix is empty, then *bes* is in fact entirely represented in the old red part and also in the new red part, qed.
assume bes_2 = []

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*

apply (*intro disjI1*)
apply (*rule-tac ?x=*res_1 **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac ?x=*rv_1 **in** *exI*)
apply (*intro conjI*)
proof −
show *subpath* (*red prb'*) *rv* res_1 rv_1 (*subs prb'*)
using *se-step*(*3*) *C* **by** (*auto simp add* : *sp-in-extends-w-subs*)
next
have $rv_1 \neq$ *tgt re* **using** *se-step*(*3*) ‹rv_1 = *src re*› **by** *auto*
thus ¬ *marked prb'* rv_1 **using** *se-step*(*3*) *B* **by** (*auto simp add* : *fringe-def*)
next
show *bes* = *ui-es* res_1 **using** *A* ‹bes_2 = []› **by** *simp*
qed

next
— If the black suffix is not empty, we first suppose that its first edge is the new edge.
assume $bes_2 \neq$ []

then obtain *be* bes_2' **where** bes_2 = *be* # bes_2' **unfolding** *neq-Nil-conv* **by** *blast*

show *?thesis*
proof (*case-tac be* = *ui-edge re*)
— If the first edge of the black suffix is represented by the new edge, then res_1 @ [*re*] is a red subpath leading to the target of the new edge, which is the fringe and not marked. Moreover, it is maximal, as there are no out-going edges from the target of the new edge in the new red part yet. Moreover, the tail of the black suffix is a suitable "new" black suffix, qed.
assume *be* = *ui-edge re*

show *?thesis*
proof (*case-tac out-edges* (*black prb*) (*fst* (*tgt re*)) = {})

assume *out-edges* (*black prb*) (*fst* (*tgt re*)) = {}

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x=*res_1@[*re*] **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac ?x=tgt re* **in** *exI*)
proof (*intro conjI*)
show *subpath* (*red prb'*) *rv* (res_1 @ [*re*]) (*tgt re*) (*subs prb'*)
using *se-step*(*3*) ‹rv_1 = *src re*› *C*
sp-in-extends-w-subs[*of re red prb red prb' rv* res_1 rv_1 *subs prb*]

```
      rb-sp-append-one[OF RB', of rv res₁ re tgt re]
    by auto
  next
    show ¬ marked prb' (tgt re)
    using se-step(3) ‹rv₁ = src re› B
    by (auto simp add : fringe-def)
  next
    have bes₂' = []
    using F ‹bes₂ = be # bes₂'›
        ‹be = ui-edge re› ‹out-edges (black prb) (fst (tgt re)) = {}›
    by (cases bes₂') (auto simp add:  Graph.sp-Cons)

    thus bes = ui-es (res₁ @ [re])
     using ‹bes = ui-es res₁ @ bes₂› ‹bes₂ = be # bes₂'› ‹be = ui-edge
re› by simp
  qed

next

  assume out-edges (black prb) (fst (tgt re)) ≠ {}

  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac ?x=res₁@[re] in exI)
  apply (rule-tac ?x=bes₂'     in exI)
  proof (intro conjI, goal-cases)
    show bes = ui-es (res₁ @ [re]) @ bes₂'
     using ‹bes = ui-es res₁ @ bes₂› ‹bes₂ = be # bes₂'› ‹be = ui-edge
re›
    by simp
  next
    case 2 show ?case
    proof (rule-tac ?x=tgt re in exI, intro conjI)
      have ¬ marked prb (src re)
      using B ‹rv₁ = src re› by (simp add : fringe-def)

      thus tgt re ∈ fringe prb'
      using se-step(3) ‹out-edges (black prb) (fst (tgt re)) ≠ {}›
            seE-fringe1[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
            seE-fringe4[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
      by auto
    next
      show subpath (red prb') rv (res₁ @ [re]) (tgt re) (subs prb')
      using se-step(3) ‹rv₁ = src re› C
          sp-in-extends-w-subs[of re red prb red prb' rv res₁ rv₁ subs prb]
            rb-sp-append-one[OF RB', of rv res₁ re tgt re]
      by auto
    next
```

show ¬ (∃ res_{21} bes_{22}. bes_2' = *ui-es* res_{21} @ bes_{22}
∧ res_{21} ≠ []
∧ *subpath-from* (*red prb'*) (*tgt re*) res_{21} (*subs prb'*))
proof (*intro notI*, *elim exE conjE*)
fix res_{21} bes_{22} rv_2

assume bes_2' = *ui-es* res_{21} @ bes_{22}
and res_{21} ≠ []
and *subpath* (*red prb'*) (*tgt re*) res_{21} rv_2 (*subs prb'*)

thus *False*
using *se-step*(*3*)
sub-rel-of.sp-from-tgt-in-extends-is-Nil
[*OF subs-sub-rel-of*[*OF se-step*(*1*)], *of re red prb'* res_{21} rv_2]
by *auto*
qed
next
show *Graph.subpath-from* (*black prb'*) (*fst* (*tgt re*)) bes_2'
using *se-step*(*3*) *F* ‹bes_2 = *be* # bes_2'› ‹*be* = *ui-edge re*›
by (*auto simp add* : *Graph.sp-Cons*)
qed
qed
qed

next
— If the first edge of the black suffix is not represented by the new edge, then this first edge is still not represented in the new red part. Hence, the source of the new edge is in the fringe of the new red part (and not marked). We conclude by showing that res_1 is a suitable red prefix, and bes_2 a suitable black suffix.
assume *be* ≠ *ui-edge re*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
apply (*intro disjI2*)
apply (*rule-tac ?x=*res_1 **in** *exI*)
apply (*rule-tac ?x=*bes_2 **in** *exI*)
apply (*intro conjI*)
apply (*rule* ‹*bes* = *ui-es* res_1 @ bes_2›)
apply (*rule-tac ?x=*rv_1 **in** *exI*)
proof (*intro conjI*)

show rv_1 ∈ *fringe prb'*
unfolding *fringe-def mem-Collect-eq*
proof (*intro conjI*)
show rv_1 ∈ *red-vertices prb'*
using *se-step*(*3*) *B* **by** (*auto simp add* : *fringe-def vertices-def*)
next
show rv_1 ∉ *subsumees* (*subs prb'*)

```
      using se-step(3) B by (auto simp add : fringe-def)
    next
      show ¬ marked prb' rv₁
      using B se-step(3) ‹rv₁ ≠ tgt re› ‹rv₁ = src re›
      by (auto simp add : fringe-def)
    next
      have be ∉ ui-edge ` out-edges (red prb') rv₁
      proof (intro notI)
        assume be ∈ ui-edge ` out-edges (red prb') rv₁

        then obtain re' where be = ui-edge re'
                        and   re' ∈ out-edges (red prb') rv₁
        by blast

        show False
        using E
        apply (elim notE)
        apply (rule-tac ?x=[re'] in exI)
        apply (rule-tac ?x=bes₂' in exI)
        proof (intro conjI)
          show bes₂ = ui-es [re'] @ bes₂'
          using ‹bes₂ = be # bes₂'› ‹be = ui-edge re'› by simp
        next
          show [re'] ≠ [] by simp
        next
          have re' ∈ edges (red prb)
          using se-step(3) ‹rv₁ = src re› ‹re' ∈ out-edges (red prb') rv₁›
                ‹be ≠ ui-edge re› ‹be = ui-edge re'›
          by (auto simp add : vertices-def)

          thus subpath-from (red prb) rv₁ [re'] (subs prb)
          using ‹re' ∈ out-edges (red prb') rv₁›
                subs-sub-rel-of[OF se-step(1)]
          by (rule-tac ?x=tgt re' in exI)
             (simp add : rb-sp-one[OF se-step(1)])
        qed
      qed

      moreover
      have be ∈ out-edges (black prb) (fst rv₁)
      using F ‹bes₂ = be # bes₂'› by (simp add : Graph.sp-Cons)

      ultimately
     show ui-edge ` out-edges (red prb') rv₁ ⊂ out-edges (black prb') (fst
rv₁)
      using se-step(3) red-OA-subset-black-OA[OF RB', of rv₁]
      by auto
    qed
  next
```

show *subpath* (*red prb'*) *rv* res_1 rv_1 (*subs prb'*)
using *se-step*(*3*) *C* **by** (*auto simp add* : *sp-in-extends-w-subs*)
next
show ¬ (∃ res_{21} bes_{22}. bes_2 = *ui-es* res_{21} @ bes_{22}
∧ res_{21} ≠ []
∧ *subpath-from* (*red prb'*) rv_1 res_{21} (*subs prb'*))
apply (*intro notI*)
apply (*elim exE conjE*)
proof −
fix res_{21} bes_{22} rv_3

assume bes_2 = *ui-es* res_{21} @ bes_{22}
and res_{21} ≠ []
and *subpath* (*red prb'*) rv_1 res_{21} rv_3 (*subs prb'*)

moreover
then obtain *re'* res_{21}' **where** res_{21} = *re'* # res_{21}'
and *be* = *ui-edge re'*
using ‹bes_2 = *be* # bes_2'› **unfolding** *neq-Nil-conv*
by (*elim exE*) *simp*

ultimately
have *re'* ∈ *edges* (*red prb'*) **by** (*simp add* : *sp-Cons*)

moreover
have *re'* ∉ *edges* (*red prb*)
using *E*
apply (*intro notI*)
apply (*elim notE*)
apply (*rule-tac ?x*=[*re'*] **in** *exI*)
apply (*rule-tac ?x*=bes_2' **in** *exI*)
proof (*intro conjI*)
show bes_2 = *ui-es* [*re'*] @ bes_2'
using ‹bes_2 = *be* # bes_2'› ‹*be* = *ui-edge re'*› **by** *simp*
next
show [*re'*] ≠ [] **by** *simp*
next
assume *re'* ∈ *edges* (*red prb*)

thus *subpath-from* (*red prb*) rv_1 [*re'*] (*subs prb*)
using *subs-sub-rel-of*[*OF se-step*(*1*)]
‹*subpath* (*red prb'*) rv_1 res_{21} rv_3 (*subs prb'*)› ‹res_{21} = *re'* #
res_{21}'›
apply (*rule-tac ?x*=*tgt re'* **in** *exI*)
apply (*simp add*: *rb-sp-Cons*[*OF RB'*])
apply (*simp add* : *rb-sp-one*[*OF se-step*(*1*)])
using *se-step*(*3*) **by** *auto*
qed

```
            ultimately
            show False
            using se-step(3) ‹be ≠ ui-edge re› ‹be = ui-edge re'›
            by auto
          qed
        next
          show Graph.subpath-from (black prb') (fst rv₁) bes₂
          using se-step(3) F by auto
        qed
      qed
    qed
  next
```

— If rv_1 is not the source of the new edge, then the out-going red edges of rv_1 in the new red part are the same as in the old red part. Thus res_1 is a suitable red prefix, and bes_2 a suitable black suffix.

```
    assume rv₁ ≠ src re

    show ?thesis
    unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
    apply (intro disjI2)
    apply (rule-tac ?x=res₁ in exI)
    apply (rule-tac ?x=bes₂ in exI)
    apply (intro conjI, goal-cases)
    proof −
      show bes = ui-es res₁ @ bes₂ by (rule ‹bes = ui-es res₁ @ bes₂›)
    next
      case 2 show ?case
      apply (rule-tac ?x=rv₁ in exI)
      apply (intro conjI, goal-cases)
      proof −

        show rv₁ ∈ fringe prb'
        using se-step(3) B ‹rv₁ ≠ src re› ‹rv₁ ≠ tgt re›
              seE-fringe1[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
              seE-fringe2[OF se-step(3)]
              seE-fringe3[OF se-step(3)]
              seE-fringe4[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
              seE-fringe5[OF se-step(3)]
        apply (case-tac marked prb (src re))

          apply simp
          apply (case-tac ui-edge ‘ out-edges (red prb') (src re) ⊂
                      out-edges (black prb) (fst (src re)))

            apply (case-tac out-edges (black prb) (fst (tgt re)) = {})
              apply simp
              apply simp

            apply (case-tac out-edges (black prb) (fst (tgt re)) = {})
```

```
      apply simp
      apply simp
    done
  next
    show subpath (red prb') rv res₁ rv₁ (subs prb')
    using se-step(3) C by (auto simp add :sp-in-extends-w-subs)
  next
    show ¬ (∃ res₂₁ bes₂₂. bes₂ = ui-es res₂₁ @ bes₂₂
                       ∧ res₂₁ ≠ []
                       ∧ subpath-from (red prb') rv₁ res₂₁ (subs prb'))
    proof (intro notI, elim exE conjE)
      fix res₂₁ bes₂₂ rv₂

      assume bes₂ = ui-es res₂₁ @ bes₂₂
      and    res₂₁ ≠ []
      and    subpath (red prb') rv₁ res₂₁ rv₂ (subs prb')

      then obtain re' res₂₁' where res₂₁ = re' # res₂₁'
      using ‹res₂₁ ≠ []› unfolding neq-Nil-conv by blast

      have rv₁ = src re' ∨ (rv₁,src re') ∈ subs prb
      and  re' ∈ edges (red prb')
      using se-step(3) rb-sp-Cons[OF RB']
          ‹subpath (red prb') rv₁ res₂₁ rv₂ (subs prb')› ‹res₂₁ = re' # res₂₁'›
      by auto

      moreover
      have re' ∈ edges (red prb)
      proof –
        have re' ≠ re
        using ‹rv₁ = src re' ∨ (rv₁,src re') ∈ subs prb›
        proof (elim disjE, goal-cases)
          case 1 thus ?thesis using ‹rv₁ ≠ src re› by auto
        next
          case 2 thus ?case using B unfolding fringe-def subsumees-conv
by fast
        qed

        thus ?thesis using se-step(3) ‹re' ∈ edges (red prb')› by simp
      qed

      show False
      using E
      apply (elim notE)
      apply (rule-tac ?x=[re'] in exI)
      apply (rule-tac ?x=ui-es res₂₁' @ bes₂₂ in exI)
      proof (intro conjI)
        show bes₂ = ui-es [re'] @ ui-es res₂₁' @ bes₂₂
        using ‹bes₂ = ui-es res₂₁ @ bes₂₂› ‹res₂₁ = re' # res₂₁'› by simp
```

next
show *[re']* ≠ [] **by** *simp*
next
show *subpath-from* (*red prb*) rv_1 [*re'*] (*subs prb*)
using *se-step*(*1*)
⟨rv_1 = *src re'* ∨ (rv_1,*src re'*) ∈ *subs prb*⟩ ⟨*re'* ∈ *edges* (*red prb*)⟩
rb-sp-one subs-sub-rel-of
by *fast*
qed
qed
next
case *4* **show** *?case* **using** *se-step*(*3*) *F* **by** *auto*
qed
qed
qed

qed

next
— If *rv* is the target of the new red edge, then we show that the empty (red) subpath is suitable prefix and *bes* a suitable suffix, using the fact that the target of the new edge is in the new fringe and can not be marked.
assume *rv* = *tgt re*

show *?thesis*
proof (*case-tac out-edges* (*black prb*) (*fst* (*tgt re*)) = {})

assume *out-edges* (*black prb*) (*fst* (*tgt re*)) = {}

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x*=[] **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac ?x*=*tgt re* **in** *exI*)
proof (*intro conjI*)
show *subpath* (*red prb'*) *rv* [] (*tgt re*) (*subs prb'*)
using *se-step*(*3*) ⟨*rv* = *tgt re*⟩ *rb-Nil-sp*[*OF RB'*] **by** (*auto simp add* : *vertices-def*)
next
have *sat* (*confs prb'* (*tgt re*))
using ⟨*bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb' rv*) (*fst rv*)⟩
⟨*rv* = *tgt re*⟩ *SE-star-sat-imp-sat*
by (*auto simp add* : *feasible-def*)

thus ¬ *marked prb'* (*tgt re*)
using *se-step*(*3*) *sat-not-marked*[*OF RB'*, *of tgt re*]
by (*auto simp add* : *vertices-def*)
next

```
    show bes = ui-es []
    using se-step(3) ‹rv = tgt re› ‹out-edges (black prb) (fst (tgt re)) = {}›
          ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)›
    by (cases bes) (auto simp add : Graph.sp-Cons)
  qed

next
  assume out-edges (black prb) (fst (tgt re)) ≠ {}

  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac ?x=[] in exI)
  apply (rule-tac ?x=bes in exI)
  proof (intro conjI, goal-cases)
    show bes = ui-es [] @ bes by simp
  next
    case 2

    show ?case
    apply (rule-tac ?x=rv in exI)
    proof (intro conjI)

      have ¬ marked prb (src re)
      proof -
        have sat (confs prb' (tgt re))
       using ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)›
             ‹rv = tgt re› SE-star-sat-imp-sat
        by (auto simp add : feasible-def)

        hence sat (confs prb' (src re))
        using se-step SE-sat-imp-sat by auto

        moreover
        have src re ≠ tgt re using se-step by auto

        ultimately
        have sat (confs prb (src re))
        using se-step(3) by (auto simp add : vertices-def)

        thus ?thesis
        using se-step sat-not-marked[OF se-step(1), of src re]
        by fast
      qed

      thus rv ∈ fringe prb'
     using se-step(3) ‹rv = tgt re› ‹out-edges (black prb) (fst (tgt re)) ≠ {}›
           seE-fringe1[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
           seE-fringe4[OF subs-sub-rel-of[OF se-step(1)]  se-step(3)]
```

```
  by auto

next

  show subpath (red prb') rv [] rv (subs prb')
  using se-step(3) ‹rv = tgt re› subs-sub-rel-of[OF RB']
  by (auto simp add : subpath-def vertices-def)

next

  show ¬ (∃ res21 bes22. bes = ui-es res21 @ bes22
                    ∧ res21 ≠ []
                    ∧ subpath-from (red prb') rv res21 (subs prb'))

  proof (intro notI, elim exE conjE)
    fix res1 bes22 rv'

    assume bes = ui-es res1 @ bes22
    and    res1 ≠ []
    and    subpath (red prb') rv res1 rv' (subs prb')

   have out-edges (red prb') (tgt re) ≠ {} ∨ tgt re ∈ subsumees (subs prb')
    proof −
      obtain re' res2 where res1 = re'#res2
      using ‹res1 ≠ []› unfolding neq-Nil-conv by blast

      hence rv = src re' ∨ (rv,src re') ∈ subs prb
      using se-step(3) ‹subpath (red prb') rv res1 rv' (subs prb')›
            rb-sp-Cons[OF RB', of rv re' res2 rv']
      by auto

      thus ?thesis
      proof (elim disjE)
        assume rv = src re'

        moreover
        hence re' ∈ out-edges (red prb') (tgt re)
        using ‹subpath (red prb') rv res1 rv' (subs prb')›
              ‹res1 = re'#res2› ‹rv = tgt re›
        by (auto simp add : sp-Cons)

        ultimately
        show ?thesis using se-step(3) by auto
      next
        assume (rv,src re') ∈ subs prb

        hence tgt re ∈ red-vertices prb
        using se-step(3) ‹rv = tgt re› subs-sub-rel-of[OF se-step(1)]
        unfolding sub-rel-of-def by force
```

```
          thus ?thesis using se-step(3) by auto
        qed
      qed

      thus False
      proof (elim disjE)
        assume out-edges (red prb') (tgt re) ≠ {}
        thus ?thesis using se-step(3)
        by (auto simp add : vertices-def image-def)
      next
        assume tgt re ∈ subsumees (subs prb')

        hence tgt re ∈ red-vertices prb
        using se-step(3) subs-sub-rel-of[OF se-step(1)]
        unfolding subsumees-conv sub-rel-of-def by fastforce

        thus ?thesis using se-step(3) by (auto simp add : vertices-def)
      qed
    qed
  next
    show Graph.subpath-from (black prb') (fst rv) bes
    using se-step(3)
        ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)›
    by simp
  qed
 qed
qed
qed
qed

next

case (mark-step prb rv₂ prb' rv₁)

have finite-RedBlack prb using mark-step by (auto simp add : finite-RedBlack-def)

show ?case
unfolding subset-iff
proof (intro allI impI)
```

— Suppose that *bes* is a (black) feasible sub-path starting at the black vertex represented by red vertex rv_1. Hence, by IH, *bes* is a red-black sub-path starting at rv_1 in the old red-black graph. We proceed by case distinction.

```
  fix bes

  assume bes ∈ feasible-subpaths-from (black prb') (confs prb' rv₁) (fst rv₁)

  then obtain c where SE-star (confs prb rv₁) (trace bes (labeling (black prb)))
c
```

and *sat c*
using *mark-step*(*3*) ‹*bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb'* rv_1) (*fst* rv_1)›
by (*simp add* : *feasible-def*) *blast*

have *bes* ∈ *RedBlack-subpaths-from prb* rv_1
using *mark-step*(*2*)[*of* rv_1] *mark-step*(*3*−*7*)
‹*bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb'* rv_1) (*fst* rv_1)›
by *auto*

thus *bes* ∈ *RedBlack-subpaths-from prb'* rv_1
apply (*subst* (*asm*) *RedBlack-subpaths-from-def*)
unfolding *Un-iff image-def Bex-def mem-Collect-eq*
proof (*elim disjE exE conjE*)
— Suppose that *bes* is entirely represented in the old red part and let us call rv_3 the red vertex where it ends. We show that it is still fully represented in the new red-part and that rv_3 is still not marked in the new red-black graph. We call *res* the red sub-path representing *bes* in the old red part.
fix *res* rv_3

assume *bes* = *ui-es res*
and *subpath* (*red prb*) rv_1 *res* rv_3 (*subs prb*)
and ¬ *marked prb* rv_3

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x=res* **in** *exI*)
proof (*intro conjI*)

show ∃ *rv'*. *subpath* (*red prb'*) rv_1 *res rv'* (*subs prb'*) ∧ ¬ *marked prb' rv'*
apply (*rule-tac ?x=*rv_3 **in** *exI*)
proof (*intro conjI*)

show *subpath* (*red prb'*) rv_1 *res* rv_3 (*subs prb'*)
using *mark-step*(*3*) ‹*subpath* (*red prb*) rv_1 *res* rv_3 (*subs prb*)›
by *auto*

next
— We then show that rv_3 is not marked.
show ¬ *marked prb'* rv_3
proof −
— *res* being a red sub-path from rv_1 to rv_3, and *c* being the configuration obtained from the configuration at rv_1 by symbolic execution of the trace of *bes* (and hence *res*), we have that *c* is subsumed by configuration at rv_3. Hence, this configuration is satisfiable, *c* being satisfiable. Thus, rv_3 can not be marked.
have *sat* (*confs prb* rv_3)
proof −
have *c* ⊑ *confs prb* rv_3

```
        using mark-step(1)
                ‹subpath (red prb) rv₁ res rv₃ (subs prb)›
                ‹bes = ui-es res›
                ‹SE-star (confs prb rv₁) (trace bes (labeling (black prb))) c›
                ‹finite-RedBlack prb›
                finite-RedBlack.SE-rel
        by simp

        thus ?thesis
        using ‹SE-star (confs prb rv₁) (trace bes (labeling (black prb))) c›
              ‹sat c›
              sat-sub-by-sat
        by fast
      qed

      thus ?thesis
      using mark-step(3) ‹subpath (red prb) rv₁ res rv₃ (subs prb)›
            lst-of-sp-is-vert[of red prb rv₁ res rv₃ subs prb]
            sat-not-marked[OF RedBlack.mark-step[OF mark-step(1,3)]]
      by auto
    qed
  qed

next
  — By construction, res represents bes.
  show bes = ui-es res by (rule ‹bes = ui-es res›)
qed

next
```

— Suppose that *bes* has a maximal red prefix res_1 leading to non-marked element rv_3 of the old fringe, and a black suffix bes_2. We show that res_1 and bes_2 are still suitable red prefix and black prefix, respectively, in the new red part.

```
  fix res₁ bes₂ rv₃ bl

  assume A : bes = ui-es res₁ @ bes₂
  and    B : rv₃ ∈ fringe prb
  and    C : subpath (red prb) rv₁ res₁ rv₃ (subs prb)
  and    D : ¬ (∃ res₂₁ bes₂₂. bes₂ = ui-es res₂₁ @ bes₂₂
                    ∧ res₂₁ ≠ []
                    ∧ subpath-from (red prb) rv₃ res₂₁ (subs prb))
  and    E : Graph.subpath (black prb) (fst rv₃) bes₂ bl

  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac ?x=res₁ in exI)
  apply (rule-tac ?x=bes₂ in exI)
  proof (intro conjI, goal-cases)
    show bes = ui-es res₁ @ bes₂ by (rule ‹bes = ui-es res₁ @ bes₂›)
```

next
case *2* **show** *?case*
apply (*rule-tac ?x=*rv_3 **in** *exI*)
proof (*intro conjI*)
— Marking a red vertex does not change the fringe, so rv_3 is in the new fringe.
have *sat* (*confs prb* rv_3)
proof −
obtain c'
where *SE-star* (*confs prb* rv_1) (*trace* (*ui-es* res_1) (*labeling* (*black prb*))) c'
and *SE-star* c' (*trace* bes_2 (*labeling* (*black prb*))) *c*
and *sat* c'
using *A* ‹*SE-star* (*confs prb* rv_1) (*trace bes* (*labeling* (*black prb*))) *c*› ‹*sat c*›
by (*simp add* : *SE-star-append SE-star-sat-imp-sat*) *blast*

moreover
hence $c' \sqsubseteq$ *confs prb* rv_3
using ‹*finite-RedBlack prb*› *mark-step*(*1*) *C finite-RedBlack.SE-rel* **by** *fast*

ultimately
show *?thesis* **by** (*simp add* : *sat-sub-by-sat*)
qed

thus $rv_3 \in$ *fringe prb'* **using** *mark-step*(*3*) *B* **by** (*auto simp add* : *fringe-def*)
next
show *subpath* (*red prb'*) rv_1 res_1 rv_3 (*subs prb'*)
using *mark-step*(*3*) ‹*subpath* (*red prb*) rv_1 res_1 rv_3 (*subs prb*)›
by *auto*
next
— We show that res_1 is a maximal prefix, which is trivial since the new red part contains less sub-paths than the old, and res_1 was already maximal.
show ¬ (∃ res_{21} bes_{22}. bes_2 = *ui-es* res_{21} @ bes_{22}
∧ $res_{21} \neq$ []
∧ *subpath-from* (*red prb'*) rv_3 res_{21} (*subs prb'*))
proof (*intro notI*, *elim exE conjE*)

fix res_{21} bes_{22} *rv4*

assume bes_2 = *ui-es* res_{21} @ bes_{22}
and $res_{21} \neq$ []
and *subpath* (*red prb'*) rv_3 res_{21} *rv4* (*subs prb'*)

show *False*
using *D*
apply (*elim notE*)

apply (*rule-tac ?x=*res_{21} **in** *exI*)
apply (*rule-tac ?x=*bes_{22} **in** *exI*)
proof (*intro conjI*)
show bes_2 = *ui-es* res_{21} @ bes_{22} **by** (*rule* ‹bes_2 = *ui-es* res_{21} @ bes_{22}›)
next
show $res_{21} \neq []$ **by** (*rule* ‹$res_{21} \neq []$›)
next
show *subpath-from* (*red prb*) rv_3 res_{21} (*subs prb*)
using *mark-step*(*3*)
‹*subpath* (*red prb'*) rv_3 res_{21} *rv4* (*subs prb'*)›
by (*simp del* : *split-paired-Ex*) *blast*
qed
qed

next
show *Graph.subpath-from* (*black prb'*) (*fst* rv_3) bes_2 **using** *mark-step*(*3*)
E **by** *simp blast*
qed
qed
qed
qed

next

case (*subsum-step prb sub prb' rv*)

hence *finite-RedBlack prb* **by** (*auto simp add* : *finite-RedBlack-def*)

have *RB'* : *RedBlack prb'* **by** (*rule RedBlack.subsum-step*[*OF subsum-step*(*1*,*3*)])

show *?case*
unfolding *subset-iff*
proof (*intro allI impI*)
— Let *bes* be a feasible sub-path starting at a black vertex represented by *rv*. By IH, *bes* is a red-black sub-path in the old red-black graph. We proceed by case distinction.
fix *bes*

assume *bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb' rv*) (*fst rv*)

hence *bes* ∈ *RedBlack-subpaths-from prb rv*
using *subsum-step*(*2*)[*of rv*] *subsum-step*(*3*−*7*) **by** *auto*

thus *bes* ∈ *RedBlack-subpaths-from prb' rv*
apply (*subst* (*asm*) *RedBlack-subpaths-from-def*)
unfolding *Un-iff image-def Bex-def mem-Collect-eq*
proof (*elim disjE exE conjE*)
— Suppose that *bes* is entirely represented in the old red part, then it is also entirely represented in the new red part, qed.

fix *res rv'*

assume *bes* = *ui-es res*
and *subpath* (*red prb*) *rv res rv'* (*subs prb*)
and ¬ *marked prb rv'*

thus *bes* ∈ *RedBlack-subpaths-from prb' rv*
using *subsum-step*(*3*) *sp-in-extends*[*of sub red prb*]
by (*simp* (*no-asm*) *only* : *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*,
intro disjI1, *rule-tac ?x=res* **in** *exI*, *intro conjI*)
(*rule-tac ?x=rv'* **in** *exI*, *auto*)

next
— Suppose that *bes* was of the form *ui-es* res_1 @ bes_2, with res_1 ending in a red vertex that we call *rv'*.
fix res_1 bes_2 *rv' bl*

assume *A* : *bes* = *ui-es* res_1 @ bes_2
and *B* : *rv'* ∈ *fringe prb*
and *C* : *subpath* (*red prb*) *rv* res_1 *rv'* (*subs prb*)
and *D* : ¬ (∃ res_{21} bes_{22}. bes_2 = *ui-es* res_{21} @ bes_{22}
∧ $res_{21} \neq []$
∧ *subpath-from* (*red prb*) *rv'* res_{21} (*subs prb*))
and *E* : *Graph.subpath* (*black prb*) (*fst rv'*) bes_2 *bl*

show *bes* ∈ *RedBlack-subpaths-from prb' rv*
proof (*case-tac rv'* = *subsumee sub*)

— Suppose that *rv'* is the newly subsumed red vertex. The idea here is to show that either bes_2 is a suitable black suffix from the new subsumer, or it is of the form *ui-es* res_{21} @ bes_{22} such that res_{21} is maximal and ends in a non-marked element of the (new) fringe, making res_1 @ res_{21} a suitable red prefix and bes_{22} a suitable black suffix for *bes* to be a red-black sub-path of the new red-black graph (note that bes_2 and bes_{22} might be empty).
However, assumptions from *subsum-step* and facts 1 to 6 are not sufficient to conclude. We proceed by backward induction on bes_2.

assume *rv'* = *subsumee sub*

show *?thesis*
using ‹*bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb' rv*) (*fst rv*)› *A C E*
proof (*induct* bes_2 *arbitrary* : *bes bl rule* : *rev-induct*, *goal-cases*)

— Suppose that the black suffix is empty, then *bes* is entirely represented by res_1 in the new red part and ends in *rv'* which is not marked, qed.

case (*1 bes bl*) **thus** *?case*

using *subsum-step*(*3*) *B sp-in-extends*[*of sub red prb*]
by (*simp* (*no-asm*) *only* :
RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq,
intro disjI1, *rule-tac ?x*=res_1 **in** *exI*, *intro conjI*)
(*rule-tac ?x*=*rv′* **in** *exI*, *auto simp add* : *fringe-def*)

next
— Suppose that the black sub-path is not empty. We call bes_2 the prefix obtained from this sub-path by removing its last edge, which we call be. We first show that *ui-es* res_1 @ bes_2 is a red-black sub-path in the old new-black graph using the "internal" induction hypothesis. We then proceed by case distinction.
case (*2 be* bes_2 *bes bl*)

then obtain c_1 c_2 c_3
where *SE-star* (*confs prb′ rv*) (*trace* (*ui-es* res_1) (*labeling* (*black prb*))) c_1
and *SE-star* c_1 (*trace* bes_2 (*labeling* (*black prb*))) c_2
and *SE* c_2 (*labeling* (*black prb*) *be*) c_3
and *sat* c_3
using *subsum-step*(*3*)
by (*simp add* : *feasible-def SE-star-append SE-star-append-one SE-star-one*) *blast*

have *ui-es* res_1 @ bes_2 ∈ *RedBlack-subpaths-from prb′ rv*
proof −
have *ui-es* res_1 @ bes_2 ∈ *feasible-subpaths-from* (*black prb′*) (*confs prb′ rv*) (*fst rv*)
proof −

have *Graph.subpath-from* (*black prb′*) (*fst rv*) (*ui-es* res_1 @ bes_2)
using *subsum-step 2*(*5*) *red-sp-imp-black-sp*[*OF subsum-step*(*1*) *C*]
by (*simp add* : *Graph.sp-append*) *blast*

moreover
have *feasible* (*confs prb′ rv*) (*trace* (*ui-es* res_1 @ bes_2) (*labeling* (*black prb′*)))
proof −
have *SE-star* (*confs prb′ rv*)
(*trace* (*ui-es* res_1@bes_2) (*labeling* (*black prb′*)))
c_2
using *subsum-step*
‹*SE-star* (*confs prb′ rv*) (*trace* (*ui-es* res_1) (*labeling* (*black prb*))) c_1›
‹*SE-star* c_1 (*trace* bes_2 (*labeling* (*black prb*))) c_2›
by (*simp add* : *SE-star-append*) *blast*

moreover
have *sat* c_2
using ‹*SE* c_2 (*labeling* (*black prb*) *be*) c_3› ‹*sat* c_3›
by (*simp add* : *SE-sat-imp-sat*)

ultimately
show *?thesis* **by** (*simp add* : *feasible-def*) *blast*
qed

ultimately
show *?thesis* **by** *simp*
qed

moreover
have *Graph.subpath-from* (*black prb*) (*fst rv'*) bes_2
using *2(5)* **by** (*auto simp add* : *Graph.sp-append-one*)

ultimately
show *?thesis* **using** *2(1,4)* **by** (*auto simp add* : *Graph.sp-append-one*)
qed

thus *?case*
apply (*subst* (*asm*) *RedBlack-subpaths-from-def*)
unfolding *Un-iff image-def Bex-def mem-Collect-eq*
proof (*elim disjE exE conjE*, *goal-cases*)
— Suppose that *ui-es* res_1 @ bes_2 is entirely represented in the new red part by a red sub-path that we call *res*, and ends in a red vertex that we call rv''. We conclude depending on the fact that be is represented by an out-going (red edge) from rv'' or not.
case (*1 res* rv'')

show *?thesis*
proof (*case-tac be* ∈ *ui-edge* ‘ *out-edges* (*red prb'*) rv'')

— If this is the case, then *bes* = *ui-es* res_1 @ bes_2 @ [*be*] is entirely represented in the new red part. We call *re* the red edge representing be from rv''. Moreover, we showed earlier that the configuration c_3 that is obtained from the configuration at *rv* by symbolic execution of (the trace of) *bes* = *ui-es* res_1 @ bes_2 @ [*be*] is satisfiable. As c_3 is subsumed by the configuration at the target of *re*, this last configuration is also satisfiable, and thus not marked, qed.

assume *be* ∈ *ui-edge* ‘ *out-edges* (*red prb'*) rv''

then obtain *re* **where** *be* = *ui-edge re*
and *re* ∈ *out-edges* (*red prb'*) rv''
by *blast*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x=res*@[*re*] **in** *exI*)
apply (*intro conjI*)

```
          apply (rule-tac ?x=tgt re in exI)
          proof (intro conjI)
            show subpath (red prb') rv (res@[re]) (tgt re) (subs prb')
        using 1(2) ‹re ∈ out-edges (red prb') rv''› by (simp add : sp-append-one)
          next
            show ¬ marked prb' (tgt re)
            proof −
              have sat (confs prb' (tgt re))
              proof −
                have subpath (red prb') rv (res@[re]) (tgt re) (subs prb')
                     using 1(2) ‹re ∈ out-edges (red prb') rv''› by (simp add :
sp-append-one)

                then obtain c
                where SE-star (confs prb' rv)
                              (trace (ui-es (res@[re])) (labeling (black prb)))
                              c
                using subsum-step(3,5,6,7) RB'
                     finite-RedBlack.sp-imp-ex-SE-star-succ[of prb' rv res@[re] tgt
re]
                unfolding finite-RedBlack-def
                by simp blast

                hence sat c
                using 1(1)
                  ‹SE-star (confs prb' rv) (trace (ui-es res1) (labeling (black prb)))
c1›
                    ‹SE-star c1 (trace bes2 (labeling (black prb))) c2›
                    ‹SE c2 (labeling (black prb) be) c3› ‹sat c3› ‹be = ui-edge re›
                    SE-star-succs-states
                  [of confs prb' rv trace (ui-es (res@[re])) (labeling (black prb)) c3]
                apply (subst (asm) eq-commute)
              by (auto simp add : SE-star-append-one SE-star-append SE-star-one
sat-eq)

                moreover
                have c ⊑ confs prb' (tgt re)
                using subsum-step(3,5,6,7) ‹subpath (red prb') rv (res@[re]) (tgt
re) (subs prb')›
                  ‹SE-star (confs prb' rv) (trace (ui-es (res@[re])) (labeling (black
prb))) c›
                  finite-RedBlack.SE-rel[of prb'] RB'
                by (simp add : finite-RedBlack-def)

                ultimately
                show ?thesis by (simp add: sat-sub-by-sat)
              qed

              thus ?thesis
```

```
              using ‹re ∈ out-edges (red prb') rv''› sat-not-marked[OF RB', of tgt re]
               by (auto simp add : vertices-def)
             qed
           next
             show bes = ui-es (res@[re]) using 1(1) 2(3) ‹be = ui-edge re› by simp
           qed

         next
```

— Suppose that *be* is not represented from *rv''*. We cannot conclude that [*be*] is a suitable suffix starting from *rv''* for proving the goal, because *rv''* might have been subsumed earlier. If this is the case, we have to show that [*be*] is a suitable suffix from the red vertex that subsumes *rv''*.

```
           assume be ∉ ui-edge ` out-edges (red prb') rv''

           show ?thesis
           proof (case-tac rv'' ∈ subsumees (subs prb'))
```

— We suppose that *rv''* is subsumed by a red vertex *arv''*. We conclude depending on the fact that be is represented in the out-going edges of *arv''* or not.

```
             assume rv'' ∈ subsumees (subs prb')

             then obtain arv'' where (rv'',arv'') ∈ (subs prb') by auto

             hence subpath (red prb') rv res arv'' (subs prb')
             using ‹subpath (red prb') rv res rv'' (subs prb')›
             by (simp add : sp-append-sub)

             show ?thesis
             proof (case-tac be ∈ ui-edge ` out-edges (red prb') arv'')
```

— If *be* is represented in the out-going edges of *arv''*, then *bes* is entirely represented in the new red part, from *rv* to *tgt re*. Moreover, the configuration at the target of *re* subsumes *c*, which is satisfiable, thus *tgt re* can not be marked, qed.

```
               assume be ∈ ui-edge ` out-edges (red prb') arv''

               then obtain re where re ∈ out-edges (red prb') arv''
                              and   be = ui-edge re
               by blast

               show ?thesis
                 unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
               apply (intro disjI1)
               apply (rule-tac ?x=res@[re] in exI)
               apply (intro conjI)
               apply (rule-tac ?x=tgt re in exI)
               proof (intro conjI)
```

show *subpath* (*red prb′*) *rv* (*res* @ [*re*]) (*tgt re*) (*subs prb′*)
using ‹*subpath* (*red prb′*) *rv res arv′′* (*subs prb′*)›
‹*re* ∈ *out-edges* (*red prb′*) *arv′′*›
by (*simp add* : *sp-append-one*)

next

have *sat* (*confs prb′* (*tgt re*))
proof −
have *subpath* (*red prb′*) *rv* (*res*@[*re*]) (*tgt re*) (*subs prb′*)
using ‹*subpath* (*red prb′*) *rv res arv′′* (*subs prb′*)›
‹*re* ∈ *out-edges* (*red prb′*) *arv′′*›
by (*simp add* : *sp-append-one*)

then obtain *c*
where *se* : *SE-star* (*confs prb′ rv*)
(*trace* (*ui-es* (*res*@[*re*])) (*labeling* (*black prb*)))
c
using *subsum-step*(*3*,*5*,*6*,*7*) *RB′*
finite-RedBlack.*sp-imp-ex-SE-star-succ*[*of prb′ rv res*@[*re*]
tgt re]
unfolding *finite-RedBlack-def*
by *simp blast*

hence *sat c*
using *1*(*1*)
‹*SE-star* (*confs prb′ rv*) (*trace* (*ui-es* res_1) (*labeling* (*black*
prb))) c_1›
‹*SE-star* c_1 (*trace* bes_2 (*labeling* (*black prb*))) c_2›
‹*SE* c_2 (*labeling* (*black prb*) *be*) c_3› ‹*sat* c_3› ‹*be* = *ui-edge re*›
SE-star-succs-states
[*of confs prb′ rv trace* (*ui-es* (*res*@[*re*])) (*labeling* (*black prb*))
c_3]
apply (*subst* (*asm*) *eq-commute*)
by (*auto simp add* : *SE-star-append-one SE-star-append SE-star-one*
sat-eq)

moreover
have *c* ⊑ *confs prb′* (*tgt re*)
using *subsum-step*(*3*,*5*,*6*,*7*) *se RB′ finite-RedBlack*.*SE-rel*[*of*
prb′]
‹*subpath* (*red prb′*) *rv* (*res*@[*re*]) (*tgt re*) (*subs prb′*)›

by (*simp add* : *finite-RedBlack-def*)

ultimately
show *?thesis* **by** (*simp add*: *sat-sub-by-sat*)
qed

thus ¬ *marked prb′* (*tgt re*)
using ‹*re* ∈ *out-edges* (*red prb′*) *arv″*›
sat-not-marked[*OF RB′*, *of tgt re*]
by (*auto simp add* : *vertices-def*)

next

show *bes* = *ui-es* (*res* @ [*re*])
using ‹*bes* = *ui-es* res_1 @ bes_2 @ [*be*]›
‹*ui-es* res_1 @ bes_2 = *ui-es res*›
‹*be* = *ui-edge re*›
by *simp*

qed

next
— Suppose that be is not represented in the out-going edges of *arv″*. We show that res is a suitable red prefix and [*be*] a suitable black prefix.
assume *A* : *be* ∉ *ui-edge* ‘ *out-edges* (*red prb′*) *arv″*

have *src be* = *fst arv″*
proof −
have *Graph.subpath* (*black prb′*) (*fst rv*) (*ui-es* res_1 @ bes_2) (*fst arv″*)
using ‹*ui-es* res_1 @ bes_2 = *ui-es res*›
‹*subpath* (*red prb′*) *rv res arv″* (*subs prb′*)›
red-sp-imp-black-sp[*OF RB′*]
by *auto*

moreover
have *Graph.subpath* (*black prb′*) (*fst rv*) (*ui-es* res_1 @ bes_2) (*src be*)
using ‹*bes* ∈ *feasible-subpaths-from* (*black prb′*) (*confs prb′ rv*) (*fst rv*)›
‹*bes* = *ui-es* res_1 @ bes_2 @ [*be*]›
by (*auto simp add* : *Graph.sp-append Graph.sp-append-one Graph.sp-one*)

ultimately
show *?thesis*
using *sp-same-src-imp-same-tgt* **by** *fast*
qed

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
apply (*intro disjI2*)
apply (*rule-tac ?x=res* **in** *exI*)
apply (*rule-tac ?x=*[*be*] **in** *exI*)

```
proof (intro conjI, goal-cases)

  show bes = ui-es res @ [be]
  using ‹bes = ui-es res1 @ bes2 @ [be]›
        ‹ui-es res1 @ bes2 = ui-es res›
  by simp

next

  case 2 show ?case
  apply (rule-tac ?x=arv'' in exI)
  proof (intro conjI)

    show arv'' ∈ fringe prb'
    unfolding fringe-def mem-Collect-eq
    proof (intro conjI)
      show arv'' ∈ red-vertices prb'
      using ‹subpath (red prb') rv res arv'' (subs prb')›
      by (simp add : lst-of-sp-is-vert)
    next
      show arv'' ∉ subsumees (subs prb')
      using ‹(rv'',arv'') ∈ subs prb'› subs-wf-sub-rel[OF RB']
      unfolding wf-sub-rel-def Ball-def
      by (force simp del : split-paired-All)
    next
      show ¬ marked prb' arv''
    using ‹(rv'',arv'') ∈ (subs prb')› subsumer-not-marked[OF RB']
      by fastforce
    next
      have be ∈ edges (black prb')
      using subsum-step(3)
            ‹Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl›
      by (simp add : Graph.sp-append-one)

        thus ui-edge ‘ out-edges (red prb') arv'' ⊂ out-edges (black
prb') (fst arv'')
      using ‹src be = fst arv''› A  red-OA-subset-black-OA[OF RB',
of arv'']

      by auto
    qed

  next

    show subpath (red prb') rv res arv'' (subs prb')
    by (rule ‹subpath (red prb') rv res arv'' (subs prb')›)

  next

    show ¬ (∃ res21 bes22. [be] = ui-es res21 @ bes22
```

```
                    ∧ res21 ≠ []
                  ∧ subpath-from (red prb') arv'' res21 (subs prb'))
  proof (intro notI, elim exE conjE, goal-cases)
    case (1 res21 bes22 rv''')

    have be ∈ ui-edge ` out-edges (red prb') arv''
    proof -
      obtain re res21' where res21 = re # res21'
      using 1(2) unfolding neq-Nil-conv by blast

      have be = ui-edge re
      and  re ∈ out-edges (red prb') arv''
      proof -
        show be = ui-edge re using 1(1) ‹res21 = re # res21'› by
simp
      next
        have re ∈ edges (red prb')
        using 1(3) ‹res21 = re # res21'› by (simp add : sp-Cons)

        moreover
        have src re = arv''
        proof -
          have (arv'',src re) ∉ subs prb'
          using ‹(rv'',arv'') ∈ subs prb'› subs-wf-sub-rel[OF RB']
          unfolding wf-sub-rel-def Ball-def
          by (force simp del : split-paired-All)

          thus ?thesis
          using 1(3) ‹res21 = re # res21'›
          by (simp add : rb-sp-Cons[OF RB'])
        qed

        ultimately
        show re ∈ out-edges (red prb') arv'' by simp
      qed

      thus ?thesis by auto
    qed

    thus False using A by (elim notE)
  qed

next

  show Graph.subpath-from (black prb') (fst arv'') [be]
  using subsum-step(3)
        ‹Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl›
        ‹(rv'',arv'') ∈ subs prb'›
        ‹subpath (red prb') rv res arv'' (subs prb')›
```

⟨*src be* = *fst arv''*⟩
RB' *red-sp-imp-black-sp subs-to-same-BL*
by (*simp add* : *Graph.sp-append-one Graph.sp-one*)
qed
qed
qed

next

— Now suppose that *rv''* is not subsumed in the new red-black graph. If *be* is represented in the out-going edges of *rv''*, then *ui-es* res_1 @ bes_2 @ [*be*] is entirely represented in the new red part. Otherwise, res is a suitable red prefix and [*be*] a suitable black prefix.

assume *rv''* ∉ *subsumees* (*subs prb'*)

show *?thesis*
proof (*case-tac be* ∈ *ui-edge* ' *out-edges* (*red prb'*) *rv''*)

assume *be* ∈ *ui-edge* ' *out-edges* (*red prb'*) *rv''*

then obtain *re* **where** *be* = *ui-edge re*
and *re* ∈ *out-edges* (*red prb'*) *rv''*
by *blast*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
apply (*intro disjI1*)
apply (*rule-tac ?x=res* @ [*re*] **in** *exI*)
apply (*intro conjI*)
apply (*rule-tac ?x=tgt re* **in** *exI*)
proof (*intro conjI*)
show *subpath* (*red prb'*) *rv* (*res* @ [*re*]) (*tgt re*) (*subs prb'*)
using ⟨*subpath* (*red prb'*) *rv res rv''* (*subs prb'*)⟩
⟨*re* ∈ *out-edges* (*red prb'*) *rv''*⟩
by (*simp add* : *sp-append-one*)
next
show ¬ *marked prb'* (*tgt re*)
proof −
have *sat* (*confs prb'* (*tgt re*))
proof −
have *subpath* (*red prb'*) *rv* (*res*@[*re*]) (*tgt re*) (*subs prb'*)
using ⟨*subpath* (*red prb'*) *rv res rv''* (*subs prb'*)⟩
⟨*re* ∈ *out-edges* (*red prb'*) *rv''*⟩
by (*simp add* : *sp-append-one*)

then obtain *c*
where *se* : *SE-star* (*confs prb' rv*)
(*trace* (*ui-es* (*res*@[*re*])) (*labeling* (*black prb*)))
c

```
                  using subsum-step(3,5,6,7) RB′
                        finite-RedBlack.sp-imp-ex-SE-star-succ[of prb′ rv res@[re]
tgt re]
                  unfolding finite-RedBlack-def
                  by simp blast

                  hence sat c
                  using 1(1)
                ‹SE-star (confs prb′ rv) (trace (ui-es res₁) (labeling (black prb)))
c₁›
                  ‹SE-star c₁ (trace bes₂ (labeling (black prb))) c₂›
                  ‹SE c₂ (labeling (black prb) be) c₃› ‹sat c₃› ‹be = ui-edge re›
                  SE-star-succs-states
                [of confs prb′ rv trace (ui-es (res@[re])) (labeling (black prb)) c₃]
                  apply (subst (asm) eq-commute)
                        by (auto simp add : SE-star-append-one SE-star-append
SE-star-one sat-eq)

                  moreover
                  have c ⊑ confs prb′ (tgt re)
                   using subsum-step(3,5,6,7) se RB′ finite-RedBlack.SE-rel[of
prb′]

                        ‹subpath (red prb′) rv (res@[re]) (tgt re) (subs prb′)›
                  by (simp add : finite-RedBlack-def)

                  ultimately
                  show ?thesis by (simp add: sat-sub-by-sat)
                qed

                thus ?thesis
               using ‹re ∈ out-edges (red prb′) rv′′› sat-not-marked[OF RB′, of
tgt re]

                by (auto simp add : vertices-def)
              qed
            next
              show bes = ui-es (res @ [re])
              using ‹bes = ui-es res₁ @ bes₂ @ [be]›
                    ‹ui-es res₁ @ bes₂ = ui-es res›
                    ‹be = ui-edge re›
              by simp
            qed

          next
            assume A : be ∉ ui-edge ‘ out-edges (red prb′) rv′′

            show ?thesis
          unfolding RedBlack-subpaths-from-def Un-iff Bex-def mem-Collect-eq
            apply (intro disjI2)
            apply (rule-tac ?x=res in exI)
```

```
apply (rule-tac ?x=[be] in exI)
proof (intro conjI, goal-cases)
  show bes = ui-es res @ [be]
  using ‹ui-es res₁ @ bes₂ = ui-es res›
        ‹bes = ui-es res₁ @ bes₂ @ [be]›
  by simp
next

  case 2

  have src be = fst rv''
  proof -
    have Graph.subpath (black prb') (fst rv) (ui-es res) (src be)
     using ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv)
(fst rv)›
       ‹bes = ui-es res₁ @ bes₂ @ [be]› ‹ui-es res₁ @ bes₂ = ui-es res›
           red-sp-imp-black-sp[OF RB' ‹subpath (red prb') rv res rv''
(subs prb')›]
   by (subst (asm)(2) eq-commute) (auto simp add : Graph.sp-append
Graph.sp-one)

    thus ?thesis
    using red-sp-imp-black-sp[OF RB' ‹subpath (red prb') rv res rv''
(subs prb')›]
    by (rule sp-same-src-imp-same-tgt)
  qed

  show ?case
  apply (rule-tac ?x=rv'' in exI)
  proof (intro conjI)

    show rv'' ∈ fringe prb'
    unfolding fringe-def mem-Collect-eq
    proof (intro conjI)
      show rv'' ∈ red-vertices prb'
      using ‹subpath (red prb') rv res rv'' (subs prb')›
      by (simp add : lst-of-sp-is-vert)
    next
      show rv'' ∉ subsumees (subs prb')
      by (rule ‹rv'' ∉ subsumees (subs prb')›)
    next
      show ¬ marked prb' rv'' by (rule ‹¬ marked prb' rv''›)
    next
      have be ∈ edges (black prb')
      using subsum-step(3)
          ‹Graph.subpath (black prb) (fst rv') (bes₂ @ [be]) bl›
      by (simp add : Graph.sp-append-one)

     thus ui-edge ‘ out-edges (red prb') rv'' ⊂ out-edges (black prb')
```

```
(fst rv'')
          using ‹src be = fst rv''› A red-OA-subset-black-OA[OF RB', of
rv']
           by auto
         qed

       next

         show subpath (red prb') rv res rv'' (subs prb')
         by (rule ‹subpath (red prb') rv res rv'' (subs prb')›)

       next

         show ¬ (∃ res21 bes22. [be] = ui-es res21 @ bes22
                           ∧ res21 ≠ []
                           ∧ SubRel.subpath-from (red prb') rv'' res21 (subs
prb'))
         proof (intro notI, elim exE conjE, goal-cases)
           case (1 res21 bes22 rv''')

           have be ∈ ui-edge ‘ out-edges (red prb') rv''
           proof −
             obtain re res21' where res21 = re # res21'
             using 1(2) unfolding neq-Nil-conv by blast

             have be = ui-edge re
             and  re ∈ out-edges (red prb') rv''
             proof −
              show be = ui-edge re using 1(1) ‹res21 = re # res21'› by
simp

             next
               have re ∈ edges (red prb')
               using 1(3) ‹res21 = re # res21'› by (simp add : sp-Cons)

               moreover
               have src re = rv''
               proof −
                 have (rv'',src re) ∉ subs prb'
                 using ‹rv'' ∉ subsumees (subs prb')›
                 by force

                 thus ?thesis
                 using 1(3) ‹res21 = re # res21'›
                 by (simp add : rb-sp-Cons[OF RB'])
               qed

               ultimately
               show re ∈ out-edges (red prb') rv'' by simp
             qed
```

thus *?thesis* **by** *auto*
qed

thus *False* **using** *A* **by** (*elim notE*)
qed

next

show *Graph.subpath-from* (*black prb'*) (*fst rv''*) [*be*]
using *subsum-step*(*3*) ‹*Graph.subpath* (*black prb*) (*fst rv'*) (bes_2 @ [*be*]) *bl*›
‹*src be* = *fst rv''*›
by (*rule-tac ?x=tgt be* **in** *exI*) (*simp add* : *Graph.sp-append-one Graph.sp-one*)

qed
qed
qed
qed
qed

next
— Now suppose that *ui-es* res_1 @ bes_2 is of the form *ui-es* $res_1{}'$ @ $bes_2{}'$. Then there exists a red vertex *rv''* such that:

- $res_1{}'$ is a maximal red prefix ending in *rv''*, which is not marked and in the new fringe,
- $bes_2{}'$ starts at the black vertex represented by *rv''*.

Note that $bes_2{}'$ can be empty. If this is the case, then we conclude depending on the fact that be is represented or not in the out-going edges of *rv''*.
If this is not the case, we show that $bes_2{}'$ @ [*be*] is also a suitable black suffix.
case (*2* $res_1{}'$ $bes_2{}'$ *rv''* *bl'*)

show *?thesis*
proof (*case-tac* $bes_2{}'$ = [])
— Suppose that $bes_2{}'$ is empty. Then either *be* is represented in the out-going edges of *rv''*, either it is not.
assume $bes_2{}'$ = []

have *Graph.subpath* (*black prb'*) (*fst rv*) (*ui-es* $res_1{}'$ @ [*be*]) *bl*
proof −
have *Graph.subpath* (*black prb'*) (*fst rv*) (*ui-es* $res_1{}'$) (*src be*)
proof −
have *Graph.subpath* (*black prb'*) (*fst rv'*) bes_2 (*src be*)
using *subsum-step*(*3*) ‹*Graph.subpath* (*black prb*) (*fst rv'*) (bes_2@[*be*]) *bl*›
by (*simp add* : *Graph.sp-append-one*)

```
      moreover
      have subpath (red prb') rv res₁ rv' (subs prb')
      using subsum-step(3) ‹subpath (red prb) rv res₁ rv' (subs prb)›
      by (auto simp add : sp-in-extends)

      hence Graph.subpath (black prb') (fst rv) (ui-es res₁) (fst rv')
      using RB' by (simp add : red-sp-imp-black-sp)

      ultimately
      show ?thesis
      using ‹ui-es res₁ @ bes₂ = ui-es res₁' @ bes₂'› ‹bes₂' = []›
      by (subst (asm) eq-commute) (auto simp add : Graph.sp-append)
    qed

    moreover
    have Graph.subpath (black prb') (src be) [be] bl
   using subsum-step(3) ‹Graph.subpath (black prb) (fst rv') (bes₂@[be])
bl›
    by (simp add : Graph.sp-append-one Graph.sp-one)

    ultimately
    show ?thesis by (auto simp add : Graph.sp-append)
  qed

  hence Graph.subpath (black prb') (fst rv) (ui-es res₁') (src be)
  and  be ∈ edges (black prb')
  and  tgt be = bl
  by (simp-all add : Graph.sp-append-one)

  have fst rv'' = src be
  proof −
    have Graph.subpath (black prb') (fst rv) (ui-es res₁') (fst rv'')
 using ‹subpath (red prb') rv res₁' rv'' (subs prb')› red-sp-imp-black-sp[OF
RB']
    by fast

    thus ?thesis
    using ‹Graph.subpath (black prb') (fst rv) (ui-es res₁') (src be)›
    by (simp add : sp-same-src-imp-same-tgt)
  qed

  show ?thesis
  proof (case-tac be ∈ ui-edge ‘ out-edges (red prb') rv'')
```

— If *be* is represented in the out-going edges of *rv''* by a red edge that we call *re*. Then *ui-es res₁' @ [be]* is entirely represented in the new red part. Moreover, the configuration at the target of *re* subsumes *c* which is satisfiable, and is in turn also satisfiable and thus not marked, qed.

```
    assume be ∈ ui-edge ‘ out-edges (red prb') rv''
```

```
then obtain re where be = ui-edge re
               and   re ∈ out-edges (red prb') rv''
by blast

show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def
mem-Collect-eq
apply (intro disjI1)
apply (rule-tac ?x=res1'@[re] in exI)
apply (intro conjI)
apply (rule-tac ?x=tgt re in exI)
proof (intro conjI)
  show subpath (red prb') rv (res1' @ [re]) (tgt re) (subs prb')
  using ‹subpath (red prb') rv res1' rv'' (subs prb')›
        ‹re ∈ out-edges (red prb') rv''›
  by (simp add : sp-append-one)
next
  show ¬ marked prb' (tgt re)
  proof -
    have sat (confs prb' (tgt re))
    proof -
      have subpath (red prb') rv (res1'@[re]) (tgt re) (subs prb')
      using ‹subpath (red prb') rv res1' rv'' (subs prb')›
            ‹re ∈ out-edges (red prb') rv''›
      by (simp add : sp-append-one)

      then obtain c
      where se : SE-star (confs prb' rv)
                     (trace (ui-es (res1'@[re])) (labeling (black prb)))
                      c
      using subsum-step(3,5,6,7) RB'
           finite-RedBlack.sp-imp-ex-SE-star-succ[of prb' rv res1'@[re]
tgt re]
      unfolding finite-RedBlack-def
      by simp blast

      hence sat c
      proof -
        have bes = ui-es (res1'@[re])
        using ‹bes = ui-es res1 @ bes2 @ [be]› ‹be = ui-edge re›
              ‹ui-es res1 @ bes2 = ui-es res1' @ bes2'› ‹bes2' = []›
        by simp

        thus ?thesis
        using subsum-step(3) SE-star-succs-states[OF se]
              ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv)
(fst rv)›
        by (auto simp add : feasible-def sat-eq)
      qed
```

```
          moreover
          have c ⊑ confs prb' (tgt re)
            using subsum-step(3,5,6,7) se finite-RedBlack.SE-rel[of prb'
RB'
                  ‹subpath (red prb') rv (res1'@[re]) (tgt re) (subs prb')›
          by (simp add : finite-RedBlack-def)

          ultimately
          show ?thesis by (simp add: sat-sub-by-sat)
        qed

        thus ?thesis
         using ‹re ∈ out-edges (red prb') rv''› sat-not-marked[OF RB', of
tgt re]
        by (auto simp add : vertices-def)
      qed
    next
      show bes = ui-es (res1' @ [re])
        using ‹bes = ui-es res1 @ bes2 @ [be]› ‹ui-es res1 @ bes2 = ui-es
res1' @ bes2'›
            ‹bes2' = []› ‹be = ui-edge re›
      by simp
    qed

  next
```

— If be is not represented in the out-going edges of rv'', then we show that $[be]$ is a suitable black suffix, $res_1{}'$ being known to be a suitable red prefix.

```
    assume A : be ∉ ui-edge ‘ out-edges (red prb') rv''

    show ?thesis
    unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
    apply (intro disjI2)
    apply (rule-tac ?x=res1' in exI)
    apply (rule-tac ?x=[be]  in exI)
    proof (intro conjI, goal-cases)
      show bes = ui-es res1' @ [be]
        using ‹bes = ui-es res1 @ bes2 @ [be]› ‹ui-es res1 @ bes2 = ui-es
res1' @ bes2'›
            ‹bes2' = []›
      by simp
    next
      case 2 show ?case
      apply (rule-tac ?x=rv'' in exI)
      proof (intro conjI)
        show rv'' ∈ fringe prb' by (rule ‹rv'' ∈ fringe prb'›)
      next
```

show *subpath* (*red prb'*) *rv* res_1' *rv''* (*subs prb'*)
by (*rule* ‹*subpath* (*red prb'*) *rv* res_1' *rv''* (*subs prb'*)›)
next
show ¬ (∃ res_{21} bes_{22}. [*be*] = *ui-es* res_{21} @ bes_{22}
∧ res_{21} ≠ []
∧ *subpath-from* (*red prb'*) *rv''* res_{21} (*subs prb'*))
proof (*intro notI*, *elim exE conjE*, *goal-cases*)
case (*1* res_{21} bes_{22} *rv'''*)

then obtain *re* res_{21}' **where** *be* = *ui-edge re*
and res_{21} = *re* # res_{21}'
unfolding *neq-Nil-conv* **by** *auto*

moreover
hence *re* ∈ *out-edges* (*red prb'*) *rv''*
using *1*(*3*) ‹*rv''* ∈ *fringe prb'*› *RB'*
unfolding *subsumees-conv* **by** (*force simp add* : *fringe-def*
rb-sp-Cons)

ultimately
show *False* **using** *A* **by** *auto*
qed
next
show *Graph.subpath-from* (*black prb'*) (*fst rv''*) [*be*]
using ‹*Graph.subpath* (*black prb'*) (*fst rv*) (*ui-es* res_1' @ [*be*]) *bl*›
‹*fst rv''* = *src be*›
by (*auto simp add* : *Graph.sp-append-one Graph.sp-one*)
qed
qed
qed

next

— Suppose that bes_2' is not empty. Then appending be at the end of bes_2' gives a suitable black suffix, qed.

assume bes_2' ≠ []

then obtain *be'* bes_2'' **where** bes_2' = *be'* # bes_2''
unfolding *neq-Nil-conv* **by** *blast*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
apply (*intro disjI2*)
apply (*rule-tac ?x=*res_1' **in** *exI*)
apply (*rule-tac ?x=*bes_2'@[*be*] **in** *exI*)
proof (*intro conjI*, *goal-cases*)
show *bes* = *ui-es* res_1' @ bes_2' @ [*be*]
using ‹*bes* = *ui-es* res_1 @ bes_2 @ [*be*]› ‹*ui-es* res_1 @ bes_2 = *ui-es* res_1'

```
@ bes2'›
              by simp
            next
              case 2 show ?case
              apply (rule-tac ?x=rv'' in exI)
              proof (intro conjI)
                show  rv'' ∈ fringe prb' by (rule ‹ rv'' ∈ fringe prb'›)
              next
                show subpath (red prb') rv res1' rv'' (subs prb')
                by (rule ‹subpath (red prb') rv res1' rv'' (subs prb')›)
              next
                show ¬ (∃ res21 bes22. bes2' @ [be] = ui-es res21 @ bes22
                                   ∧ res21 ≠ []
                                   ∧ subpath-from (red prb') rv'' res21 (subs prb'))
                proof (intro notI, elim exE conjE, goal-cases)
                  case (1 res21 bes22 rv''')

                  then obtain re res21' where res21 = re # res21'
                                        and   be' = ui-edge re
                  using ‹bes2' = be' # bes2''› unfolding neq-Nil-conv by auto

                  show False
                  using ‹¬ (∃ res21 bes22. bes2' = ui-es res21 @ bes22
                                        ∧ res21 ≠ []
                                       ∧ subpath-from (red prb') rv'' res21 (subs prb'))›
                  apply (elim notE)
                  apply (rule-tac ?x=[re] in exI)
                  apply (rule-tac ?x=bes2'' in exI)
                  proof (intro conjI)
                    show bes2' = ui-es [re] @ bes2''
                    using ‹bes2' @ [be] = ui-es res21 @ bes22› ‹bes2' = be' # bes2''›
                          ‹be' = ui-edge re›
                    by simp
                  next
                    show [re] ≠ [] by simp
                  next
                    show subpath-from (red prb') rv'' [re] (subs prb')
                    using ‹subpath (red prb') rv'' res21 rv''' (subs prb')› ‹res21 = re
# res21'›
                    by (fastforce simp add : sp-Cons Nil-sp vertices-def)
                  qed
                qed

              next

                show Graph.subpath-from (black prb') (fst rv'') (bes2' @ [be])
                proof −
                  have Graph.subpath (black prb') (fst rv) (ui-es res1' @ bes2') (src
be)
```

proof −
have *Graph.subpath* (*black prb*′) (*fst rv*) (*ui-es res*$_1$ @ *bes*$_2$) (*src be*)
using ‹*bes* ∈ *feasible-subpaths-from* (*black prb*′) (*confs prb*′ *rv*) (*fst rv*)›
‹*bes* = *ui-es res*$_1$ @ *bes*$_2$ @ [*be*]›
by (*auto simp add* : *Graph.sp-append Graph.sp-one*)

thus *?thesis* **using** ‹*ui-es res*$_1$ @ *bes*$_2$ = *ui-es res*$_1$′@*bes*$_2$′› **by** *simp*
qed

moreover
have *Graph.subpath* (*black prb*′) (*fst rv*) (*ui-es res*$_1$′ @ *bes*$_2$′) *bl*′
using ‹*Graph.subpath* (*black prb*′) (*fst rv*′′) *bes*$_2$′ *bl*′›
red-sp-imp-black-sp[*OF RB*′ ‹*subpath* (*red prb*′) *rv res*$_1$′ *rv*′′ (*subs prb*′)›]
by (*auto simp add* : *Graph.sp-append*)

ultimately
have *src be* = *bl*′ **by** (*rule sp-same-src-imp-same-tgt*)

moreover
have *Graph.subpath* (*black prb*′) (*src be*) [*be*] (*tgt be*)
using *subsum-step*(*3*) ‹*Graph.subpath* (*black prb*) (*fst rv*′) (*bes*$_2$@[*be*]) *bl*›
by (*auto simp add* : *Graph.sp-append-one Graph.sp-one*)

ultimately
show *?thesis*
using ‹*Graph.subpath* (*black prb*′) (*fst rv*′′) *bes*$_2$′ *bl*′›
by (*simp add* : *Graph.sp-append-one Graph.sp-one*)
qed
qed
qed
qed
qed
qed

next

— Suppose that *rv*′ is not the newly subsumed red vertex. Hence, *rv*′ is still not marked and in the fringe and *res*$_1$ is still maximal, which makes *res*$_1$ and *bes*$_2$ suitable red prefix and black suffix in the new red part.

assume *rv*′ ≠ *subsumee sub*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*

```
apply (intro disjI2)
apply (rule-tac ?x=res₁ in exI)
apply (rule-tac ?x=bes₂ in exI)
proof (intro conjI, goal-cases)
  show bes = ui-es res₁ @ bes₂ by (rule ‹bes = ui-es res₁ @ bes₂›)
next

  case 2 show ?case
  apply (rule-tac ?x=rv' in exI)
  proof (intro conjI)
    show rv'∈ fringe prb'
      using subsum-step(3) subsumE-fringe[OF subsum-step(3)] B ‹rv' ≠
subsumee sub›
    by simp
  next
    show subpath (red prb') rv res₁ rv' (subs prb')
    using subsum-step(3) C by (auto simp add : sp-in-extends)
  next
    show ¬ (∃ res₂₁ bes₂₂. bes₂ = ui-es res₂₁ @ bes₂₂
                    ∧ res₂₁ ≠ []
                    ∧ subpath-from (red prb') rv' res₂₁ (subs prb'))
    proof (intro notI, elim exE conjE)
      fix res₂₁ bes₂₂ rv''

      assume bes₂ = ui-es res₂₁ @ bes₂₂
      and    res₂₁ ≠ []
      and    subpath (red prb') rv' res₂₁ rv'' (subs prb')

      then obtain re res₂₁' where res₂₁ = re # res₂₁'
      unfolding neq-Nil-conv by blast

      have subpath (red prb) rv' [re] (tgt re) (subs prb)
      proof -
        have ¬ uses-sub rv' [re] (tgt re) sub using ‹rv' ≠ subsumee sub› by
auto

        thus ?thesis
        using subsum-step(3)
          ‹subpath (red prb') rv' res₂₁ rv'' (subs prb')› ‹res₂₁ = re # res₂₁'›
          sp-in-extends-not-using-sub
          rb-sp-Cons[OF RB', of rv' re res₂₁' rv'']
          rb-sp-one[OF subsum-step(1), of rv' re tgt re]
          subs-sub-rel-of[OF subsum-step(1)]
        by auto
      qed

      show False
      using D
      apply (elim notE)
```

apply (*rule-tac ?x=*[*re*] **in** *exI*)
apply (*rule-tac ?x=ui-es* res_{21}*'@*bes_{22} **in** *exI*)
proof (*intro conjI*)
show bes_2 = *ui-es* [*re*] @ *ui-es* res_{21}' @ bes_{22}
using ‹bes_2 = *ui-es* res_{21} @ bes_{22}› ‹res_{21} = *re* # res_{21}'› **by** *simp*
next
show [*re*] ≠ [] **by** *simp*
next
show *subpath-from* (*red prb*) *rv'* [*re*] (*subs prb*)
apply (*rule-tac ?x=tgt re* **in** *exI*)
using *subsum-step*(*3*)
‹*rv'* ≠ *subsumee sub*› ‹*subpath* (*red prb'*) *rv'* res_{21} *rv''* (*subs prb'*)›
‹res_{21} = *re* # res_{21}'›
rb-sp-Cons[*OF RB'*, *of rv' re* res_{21}' *rv''*]
rb-sp-one[*OF subsum-step*(*1*), *of rv' re tgt re*]
subs-sub-rel-of[*OF subsum-step*(*1*)] *subs-sub-rel-of*[*OF RB'*]
by *fastforce*
qed
qed
next
show *Graph.subpath-from* (*black prb'*) (*fst rv'*) bes_2
using *subsum-step*(*3*) *E* **by** *simp blast*
qed
qed
qed
qed
qed

next

case (*abstract-step prb* rv_2 c_a *prb'* rv_1)

have *RB'* : *RedBlack prb'* **by** (*rule RedBlack.abstract-step*[*OF abstract-step*(*1*,*3*)])
have *finite-RedBlack prb* **using** *abstract-step* **by** (*auto simp add* : *finite-RedBlack-def*)

show *?case*
unfolding *subset-iff*
proof (*intro allI impI*)
— Suppose that *bes* is a feasible sub-path starting at the black vertex represented by the red vertex rv_1. We proceed depending on the fact that rv_1 is the red vertex where the abstraction took place or not. We have to make this distinction to be able to use our IH, in the case where @$rv_1 \neq rv_2$.
fix *bes*

assume *bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb'* rv_1) (*fst* rv_1)

show *bes* ∈ *RedBlack-subpaths-from prb'* rv_1
proof (*case-tac* rv_2 = rv_1)
— If this is the case, then the only possible red prefix is the empty edge sequence.

By definition of the abstraction operator, we have that the empty sequence is indeed a suitable red prefix and that *bes* is suitable black prefix.

```
assume rv₂ = rv₁

show ?thesis
proof (case-tac out-edges (black prb') (fst rv₁) = {})
  assume out-edges (black prb') (fst rv₁) = {}

  show ?thesis
 unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
  apply (intro disjI1)
  apply (rule-tac ?x=[] in exI)
  apply (intro conjI)
  apply (rule-tac ?x=rv₁ in exI)
  proof (intro conjI)
    show subpath (red prb') rv₁ [] rv₁ (subs prb')
    using abstract-step(4) rb-Nil-sp[OF RB'] by fast
  next
    show ¬ marked prb' rv₁ using abstract-step(3) ‹rv₂ = rv₁› by simp
  next
    show bes = ui-es []
    using ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv₁) (fst rv₁)›
          ‹out-edges (black prb') (fst rv₁) = {}›
    by (cases bes) (auto simp add : Graph.sp-Cons)
  qed

next
  assume out-edges (black prb') (fst rv₁) ≠ {}

  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac ?x=[]  in exI)
  apply (rule-tac ?x=bes in exI)
  proof (intro conjI, goal-cases)

    show bes = ui-es [] @ bes by simp

  next

    case 2 show ?case
    apply (rule-tac ?x=rv₁ in exI)
    proof (intro conjI)

      show rv₁ ∈ fringe prb'
       using abstract-step(1,3) ‹rv₂ = rv₁› ‹out-edges (black prb') (fst rv₁) ≠
{}›
      by (auto simp add : fringe-def)
```

next

show *subpath* (*red prb'*) rv_1 [] rv_1 (*subs prb'*)
using *abstract-step*(*3*) ‹$rv_2 = rv_1$›
rb-Nil-sp[*OF RedBlack.abstract-step*[*OF abstract-step*(*1*,*3*)]]
by *auto*

next

show ¬ (∃ res_{21} bes_{22}. *bes* = *ui-es* res_{21} @ bes_{22}
∧ $res_{21} \neq$ []
∧ *subpath-from* (*red prb'*) rv_1 res_{21} (*subs prb'*))
proof (*intro notI*, *elim exE conjE*)
fix res_{21} rv_3

assume $res_{21} \neq$ []
and *subpath* (*red prb'*) rv_1 res_{21} rv_3 (*subs prb'*)

moreover
then obtain *re* res_{21}' **where** res_{21} = *re* # res_{21}' **unfolding** *neq-Nil-conv*
by *blast*

ultimately
have *re* ∈ *out-edges* (*red prb'*) rv_1
using *abstract-step*(*3*) ‹$rv_2 = rv_1$›
rb-sp-Cons[*OF RedBlack.abstract-step*[*OF abstract-step*(*1*,*3*)], *of*
rv_1 *re* res_{21}' rv_3]
unfolding *subsumees-conv* **by** *fastforce*

thus *False* **using** *abstract-step*(*3*) ‹$rv_2 = rv_1$› **by** *auto*
qed

next

show *Graph.subpath-from* (*black prb'*) (*fst* rv_1) *bes*
using ‹*bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb'* rv_1) (*fst* rv_1)›
by *simp*

qed
qed
qed

next
— Suppose that rv_1 is not the red vertex where the abstraction took place. Then, as abstracting a configuration has no effect on the rest of the red tree, we can show by IH that *bes* is red-black sub-path of the old red-black graph. We conclude by case distinction.
assume $rv_2 \neq rv_1$

```
moreover
hence feasible (confs prb rv₁) (trace bes (labeling (black prb)))
using abstract-step(3)
      ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv₁) (fst rv₁)›
by simp

ultimately
have bes ∈ RedBlack-subpaths-from prb rv₁
using abstract-step(2)[of rv₁] abstract-step(3−7)
      ‹bes ∈ feasible-subpaths-from (black prb') (confs prb' rv₁) (fst rv₁)›
by auto

thus ?thesis
apply (subst (asm) RedBlack-subpaths-from-def)
unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

  fix res rv₃

  assume  bes = ui-es res
  and     subpath (red prb) rv₁ res rv₃ (subs prb)
  and     ¬ marked prb rv₃

  thus ?thesis
  using abstract-step(3)
unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
  by (intro disjI1, rule-tac ?x=res in exI, intro conjI)
     (rule-tac ?x=rv₃ in exI, simp-all)
next

  fix res₁ bes₂ rv₃ bl

  assume A : bes = ui-es res₁ @ bes₂
  and    B : rv₃ ∈ fringe prb
  and    C : subpath (red prb) rv₁ res₁ rv₃ (subs prb)

  and    E : ¬ (∃ res₂₁ bes₂₂. bes₂ = ui-es res₂₁ @ bes₂₂
                      ∧ res₂₁ ≠ []
                      ∧ subpath-from (red prb) rv₃ res₂₁ (subs prb))
  and    F : Graph.subpath (black prb) (fst rv₃) bes₂ bl

  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac ?x=res₁ in exI)
  apply (rule-tac ?x=bes₂ in exI)
  proof (intro conjI, goal-cases)
    show bes = ui-es res₁ @ bes₂ by (rule ‹bes = ui-es res₁ @ bes₂›)
  next
```

case *2* **show** *?case*
using *abstract-step*(*3*) *B C E F* **unfolding** *fringe-def*
by (*rule-tac ?x*=rv_3 **in** *exI*) *auto*
qed
qed
qed
qed

next
— Strengthening a configuration with an invariant will help refuse "brutal" abstractions. As all abstractions preserves the set of feasible paths, we conclude.

case (*strengthen-step prb* rv_2 *e prb'* rv_1)

show *?case*
unfolding *subset-iff*
proof (*intro allI impI*)

fix *bes*

assume *bes* ∈ *feasible-subpaths-from* (*black prb'*) (*confs prb'* rv_1) (*fst* rv_1)

hence *bes* ∈ *RedBlack-subpaths-from prb* rv_1
using *strengthen-step*(*2*)[*of* rv_1] *strengthen-step*(*3*−*7*) **by** *auto*

thus *bes* ∈ *RedBlack-subpaths-from prb'* rv_1
apply (*subst* (*asm*) *RedBlack-subpaths-from-def*)
unfolding *Un-iff image-def Bex-def mem-Collect-eq*
proof (*elim disjE exE conjE*)

fix *res* rv_2

assume *bes* = *ui-es res*
and *subpath* (*red prb*) rv_1 *res* rv_2 (*subs prb*)
and ¬ *marked prb* rv_2

thus *?thesis*
using *strengthen-step*(*3*)
unfolding *RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq*
by (*intro disjI1*) *fastforce*

next

fix res_1 bes_2 rv_3 *bl*

assume *A* : *bes* = *ui-es* res_1 @ bes_2
and *B* : rv_3 ∈ *fringe prb*
and *C* : *subpath* (*red prb*) rv_1 res_1 rv_3 (*subs prb*)

and E : ¬ (∃ res_{21} bes_{22}. bes_2 = *ui-es* res_{21} @ bes_{22}
∧ res_{21} ≠ []
∧ *subpath-from* (*red prb*) rv_3 res_{21} (*subs prb*))
and F : *Graph.subpath* (*black prb*) (*fst* rv_3) bes_2 *bl*

show *?thesis*
unfolding *RedBlack-subpaths-from-def Un-iff mem-Collect-eq*
apply (*intro disjI2*)
apply (*rule-tac ?x=*res_1 **in** *exI*)
apply (*rule-tac ?x=*bes_2 **in** *exI*)
proof (*intro conjI*, *goal-cases*)
show *bes* = *ui-es* res_1 @ bes_2 **by** (*rule* ‹*bes* = *ui-es* res_1 @ bes_2›)
next
case *2* **show** *?case*
using *strengthen-step*(*3*) *B C E F* **unfolding** *fringe-def* **by** (*rule-tac ?x=*rv_3 **in** *exI*) *auto*
qed

qed
qed
qed

Red-black paths being red-black sub-path starting from the red root, and feasible paths being feasible sub-paths starting at the black initial location, it follows from the previous theorem that the set of feasible paths when considering the configuration of the root is a subset of the set of red-black paths.

theorem (**in** *finite-RedBlack*)
assumes *RedBlack prb*
shows *feasible-paths* (*black prb*) (*confs prb* (*root* (*red prb*))) ⊆ *RedBlack-paths prb*
using *feasible-subpaths-preserved*[*OF assms*, *of root* (*red prb*)] *consistent-roots*[*OF assms*]
by (*simp add* : *vertices-def*)

The configuration at the red root might have been abstracted. In this case, the initial configuration is subsumed by the current configuration at the root. Thus the set of feasible paths when considering the initial configuration is also a subset of the set of red-black paths.

lemma *init-subsumed* :
assumes *RedBlack prb*
shows *init-conf prb* ⊑ *confs prb* (*root* (*red prb*))
using *assms*
proof (*induct prb*)
case *base* **thus** *?case* **by** (*simp add*: *subsums-refl*)
next
case *se-step* **thus** *?case* **by** (*force simp add* : *vertices-def*)
next
case *mark-step* **thus** *?case* **by** *simp*

```
next
  case subsum-step thus ?case by simp
next
  case (abstract-step prb rv c_a prb')
  thus ?case by (auto simp add : subsums-trans abstract-def)
next
  case strengthen-step thus ?case by simp
qed


lemma (in finite-RedBlack)
  assumes RedBlack prb
  shows   feasible-paths (black prb) (init-conf prb) ⊆ RedBlack-paths prb
unfolding subset-iff mem-Collect-eq
proof (intro allI impI, elim exE conjE, goal-cases)
  case (1 es bl)

  hence es ∈ feasible-subpaths-from (black prb) (init-conf prb) (fst (root (red prb)))
  using consistent-roots[OF assms] by simp blast

  hence es ∈ feasible-subpaths-from (black prb) (confs prb (root (red prb))) (fst
(root (red prb)))
  unfolding mem-Collect-eq
  proof (elim exE conjE, goal-cases)
    case (1 bl')

    show ?case
    proof (rule-tac ?x=bl' in exI, intro conjI)
      show Graph.subpath (black prb) (fst (root (red prb))) es bl' by (rule 1(1))
    next
      have finite-labels (trace es (labeling (black prb)))
      using finite-RedBlack by auto

      moreover
      have finite (pred (confs prb (root (red prb))))
      using finite-RedBlack finite-pred[OF assms]
      by (auto simp add : vertices-def finite-RedBlack-def)

      moreover
      have finite (pred (init-conf prb))
      using assms by (intro finite-init-pred)

      moreover
      have ∀ e∈pred (confs prb (root (red prb))). finite (Bexp.vars e)
      using finite-RedBlack finite-pred-constr-symvars[OF assms]
      by (fastforce simp add : finite-RedBlack-def vertices-def)

      moreover
      have ∀ e∈pred (init-conf prb). finite (Bexp.vars e)
```

```
      using assms by (intro finite-init-pred-symvars)

      moreover
      have init-conf prb ⊑ confs prb (root (red prb))
      using assms by (rule init-subsumed)

      ultimately
      show feasible (confs prb (root (red prb))) (trace es (labeling (black prb)))
      using 1(2) by (rule subsums-imp-feasible)
    qed
  qed

  thus ?case
  using feasible-subpaths-preserved[OF assms, of root (red prb)]
  by (auto simp add : vertices-def)
qed

end
```

A.13 Conclusion

We have formally proved the correctness of a set of graph transformations used by systems that compute approximations of sets of (feasible) paths by building symbolic evaluation graphs with unbounded loops. Formalizing all the details needed for a machine-checked proof was a substantial piece of work. To our knowledge, such formalization was not done before.

The ATRACER model separates the fundamental aspects and the heuristic parts of the algorithm. Additional graph transformations for restricting abstractions or for computing interpolants or invariants can be added to the current framework, reusing the existing machinery for graphs, paths, configurations, etc.

FiXme Note: ATRACER not defined

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Détection de Chemins Infaisables : un Modèle Formel et un Algorithme

Mots clefs : Test de logiciels, génération aléatoire, exécution symbolique, chemins infaisables, méthodes formelles

Résumé : De nombreuses techniques d'analyse de programmes se basent sur une représentation du code sous forme de graphe, appelée Graphe de Flot de Contrôle (CFG). Un CFG est une représentation compacte du comportement d'un programme : chaque comportement est représenté par exactement un chemin dans le CFG. La propriété inverse n'est pas vraie : chaque chemin du CFG ne représente pas nécessairement un comportement du programme. Les CFG sont des sur-approximations imprécises des ensembles d'exécutions des programmes. Les chemins ne représentant pas une possible exécution du programme sont dits infaisables, puisque le programme ne peut jamais s'exécuter le long d'un tel chemin. En général, le nombre de chemins infaisables surpasse largement celui des faisables, même pour des programmes simples. Par conséquent, les techniques basées sur les CFG – test boîte blanche, model checking, analyse statique, par exemple – sont négativement impactées par l'existence des chemins infaisables.
Les travaux présentés dans ce document concernent l'élimination de chemins infaisables dans des CFG. Pour ce faire, nous proposons un algorithme basé sur l'exécution symbolique et la détection de subsomptions et dirigé par diverses heuristiques. Cet algorithme peut être vu comme un dépliage potentiellement partiel du CFG de départ durant lequel des chemins infaisables sont éliminés. Cet algorithme est basé sur un modèle formel montrant la correction de l'approche et le fait que les chemins faisables du graphe de départ soient conservés.

Title : Infeasible Paths Detection: a Formal Model and an Algorithm

Keywords : Software testing, random generation, symbolic execution, infeasible paths, formal methods

Abstract : A number of program analysis techniques are based on a graphical representation of the program called the Control Flow Graph (CFG). A CFG is a compact representation of a program's behavior: each possible execution of the program is represented by exactly one path in the CFG. The inverse property is not true: not every path of the CFG represents an actual execution of the program. As a result, CFG are inaccurate over-approximations of the sets of executions of programs. Paths that do not represent actual executions are said to be infeasible since the program can never run along one of them. In general, the infeasible paths largely outnumber the feasible ones, even for simple programs. As a result, techniques based on CFG – white-box testing, model checking, static analysis, for example – are negatively impacted by the existence of infeasible paths.
The works presented in this document focus on pruning infeasible paths from CFG. To do so, we propose an algorithm based on symbolic execution and detection of subsumptions and driven by various heuristics. It can be seen as a potentially partial unfolding of the original CFG during which infeasible paths are pruned. This algorithm is based on a formal model that shows the correction of the approach and the fact that it preserves the feasible paths of the input graph.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France