Program Analysis with Interpolants

Georg Weißenbacher
Magdalen College

A thesis submitted for the degree of
Doctor of Philosophy

Trinity Term, 2010

Abstract

This dissertation discusses novel techniques for interpolation-based software model checking, an approximate method which uses Craig interpolation to compute invariants of programs. Our work addresses two aspects of program analyses based on model checking: verification (the construction of correctness proofs for programs) and falsification (the detection of counterexamples that violate the specification).

In Hoare's calculus, a proof of correctness comprises assertions which establish that a program adheres to its specification. The principal challenge is to derive appropriate assertions and loop invariants. Contemporary software verification tools use Craig interpolation (as opposed to traditional predicate transformers such as the weakest precondition) to derive approximate assertions. The performance of the model checker is contingent on the Craig interpolants computed.

We present novel interpolation techniques which provide the following advantages over existing methods. Firstly, the resulting interpolants are sound with respect to the bit-level semantics of programs, which is an improvement over interpolation systems that use linear arithmetic over the reals to approximate bit-vector arithmetic and/or do not support bit-level operations. Secondly, our interpolation systems afford us a choice of interpolants and enable us to fine-tune their logical strength and structure. In contrast, existing procedures are limited to a single *ad hoc* choice of an interpolant.

Interpolation-based verification tools are typically forced to refine an initial approximation repeatedly in order to achieve the accuracy required to establish or refute the correctness of a program. The detection of a counterexample containing a repetitive construct may necessitate one refinement step (involving the computation of additional interpolants) for each iteration of the loop. We present a heuristic that aims to avoid the repeated and computationally expensive construction of interpolants, thus enabling the detection of deeply buried defects such as buffer overflows.

Finally, we present an implementation of our techniques and evaluate them on a set of standardised device driver and buffer overflow benchmarks.

Declaration of Originality

I, Georg Weißenbacher, do herewith declare that the material contained in this dissertation is original work performed by me under the guidance of my supervisor Dr. Daniel Kröning and to the best of my knowledge contains no work previously published or written by another person, except where due acknowledgement is made.

Acknowledgements

A D.Phil. dissertation, though written (and often also read) by only a single person, invariably bears the marks of its creator's intellectual environment. Consequently, I am indebted to a number of brilliant people who supported me in my academic endeavours. My interest in formal verification was sparked by Peter Lucas and Bernhard Aichernig, who, in my undergraduate years at Graz University of Technology, introduced me to Hoare Logic. Little did I know at the time that several years later this formalism would play a pivotal role in my doctoral research, and that I was to find myself in a position to discuss my work with its inventor, Sir Tony Hoare. This encounter as well as my doctoral studies were made possible by a generous scholarship provided by Microsoft Research Cambridge.

I thoroughly enjoyed working with Roderick Bloem, who supervised my master's thesis and had a lasting impact on my academic career. He suggested that I work on software model checking and predicate abstraction, a research direction which I continued pursuing during an internship at Microsoft Research in Redmond, also made possible by Roderick. During my time at Microsoft Research, I was given the honour to work with and learn from the ingenious researchers Sriram Rajamani, Thomas Ball, and Byron Cook. It was the latter who, a few years later, introduced me to Daniel Kroening, whose offer to join his group at ETH Zürich made me return to academia after spending some time in industry.

Working under Daniel Kroening's supervision was a remarkable and rewarding experience. What I learned from him went far beyond an ordinary doctoral education. The time spent as a member of his group was a holistic apprenticeship in the academic trade. To him I owe, among many other things, the opportunity to co-author a textbook (a rare experience for a doctoral student) and to study in two very different academic environments in Zürich and Oxford.

The talented students and researchers in Daniel's group were another reason why I enjoyed my time as a doctoral student so much. We soon became not only collaborators but close friends. My fellow doctoral students Gérard Basler, Nicolas Blanc, Angelo Brillout, Yury Chebiryak, Vijay D'Silva, Leopold Haller, Alexander Kaiser, Mitra Purandare, and Christoph Wintersteiger, and the post-doctoral researchers Alastair Donaldson, Nannan He, Philipp Rümmer, and Thomas Wahl have helped me with my research and enriched my student life in more ways than I can count. I want to express special gratitude to my colleagues Christoph, Gérard, Mitra, Nicolas, and Mark Kattenbelt for their advice and help with my implementation and experimental evaluation, to Vijay for the enlightening lessons in abstract interpretation, and to Alastair, Philipp, and Thomas for the insightful comments on my dissertation.

It is with gratitude and pride that I thank my illustrious examiners Thomas Henzinger and Tom Melham for their willingness to sacrifice their time to assess my work. A doctoral candidate cannot wish for a more thorough and profound review of his achievements.

I want to express my deepest gratitude to the wonderful May Chan, who made my life complete (and also helped me to eliminate a number of linguistic singularities from my thesis). Finally, I am grateful to my parents for their continuous and loving support throughout my academic odyssey.

Contents

1 Introduction **1**

1.1 A Brief History of Automated Software Verification 1
1.2 A Mission Statement 6
1.3 Two Motivating Examples 7
1.3.1 Verification 8
1.3.2 Falsification 14
1.4 Contributions 17
1.5 Significance of Contributions 20
1.6 Organisation of this Dissertation 21

2 Interpolation-based Verification Techniques **23**

2.1 Hoare Logic and Dijkstra's Predicate Transformers 24
2.2 Craig Interpolation 31
2.3 Programs, Reachability Trees, and Counterexamples 39
2.4 Predicate Abstraction 43
2.4.1 Approximating Predicate Transformers 43
2.4.2 Abstract Domains and Transition Relations 47
2.4.3 Boolean Programs 53
2.5 Construction of Safety Proofs 55
2.5.1 Complete Reachability Trees and Inductive Invariants 56
2.5.2 Inductive Invariants and Safety Proofs 59
2.5.3 Refining Approximations 61
2.5.4 Counterexample-Guided Abstraction Refinement 65
2.6 Verification Tools and Related Work 68

3 Constructing Interpolants **71**

3.1 Bit-Vector Arithmetic 72
3.2 Interpolation, Symbol Elimination, and Quantification 77
3.2.1 Proofs and Interpolants for Bit-Vector Arithmetic 77
3.2.2 Local Derivations and Symbol Elimination 79
3.2.3 Partial Interpolants 80
3.2.4 Symbol and Quantifier Elimination 83
3.3 Interpolation and Propositional Logic 87
3.3.1 Flattening Bit-Vector Formulae 88
3.3.2 Resolution Refutations 93
3.3.3 Interpolation Systems for Propositional Resolution Proofs 95
3.3.4 Modulating Interpolant Strength 110

3.3.5 Proof Transformations and Interpolant Strength 113
3.4 Propositional Proofs and Bit-Vector Arithmetic 120
3.4.1 SMT-Solvers and Blocking Clauses 122
3.4.2 Lifting Propositional Resolution Proofs to the Word-Level 125
3.5 Interpolation for Word-Level Bit-Vector Arithmetic 133
3.5.1 Inference Rules for Bit-Vector Arithmetic 133
3.5.2 Extracting Interpolants from Local Refutations 137
3.6 Experimental Results 145
3.7 Related Work 153

4 Counterexamples and Refinement with Loops 155
4.1 Counterexamples with Loops 157
4.1.1 How Predicate Abstraction Handles Loops 157
4.1.2 Detecting Loops in Abstract Counterexamples 159
4.1.3 Checking the Safety of Counterexamples with Loops 160
4.2 Refinement in the Presence of Loops 166
4.2.1 Refinement Using Closed Recurrence Equations 166
4.2.2 Refinement Using Unwound Spurious Counterexamples 170
4.3 Examples 171
4.4 Conditions for Completeness 174
4.5 Experimental Results 177
4.6 Related Work 180

5 Future Directions and Conclusion 185
5.1 Open Issues 187
5.2 Future Directions 187
5.3 Conclusion 189

A Implementation 191
A.1 Implementation Details of Wolverine 191
A.2 A Graph-Based Decision Procedure 192

B Proofs 205
B.1 Proofs for Section 2.4 205
B.2 Proofs for Section 3.3.3 207
B.3 Proofs for Section 3.3.4 213
B.4 Proofs for Section 3.3.5 217
B.5 Proofs for Section 3.5.2 223

Bibliography 227

Chapter 1

Introduction

This dissertation is concerned with the automated verification of software. The main focus of our work is interpolation-based model checking, an approximate method for computing invariants of symbolic transition systems.

We start this chapter with a brief overview of the history of automated software verification. This historical account explains the motivation of our work and introduces the specific techniques which we discuss in this dissertation.

We then present our mission statement and identify the specific class of verification problems we address. This is followed by a discussion of two instances of such problems, serving as motivating examples. We then summarise our contributions and present an outline of our thesis.

1.1 A Brief History of Automated Software Verification

We do not make the bold claim that the following narration is an objective or holistic account of the historic events in the field of software verification. It ignores entire lines of research, such as type theory and theorem proving, and only mentions others, such as static analysis and abstract interpretation, in passing. Our intention is to put the contribution of this thesis into an historical perspective and context. To this aim, we recap the story leading up to predicate abstraction and interpolation-based model checking.

Furthermore, as befits a proper tale, it is passed on and modified in the process of doing

so. The first part of our historic account is loosely based on Vijay Victor D'Silva's talk "Tales from Verification History".[1]

The beginning of the history of automated software verification notably predates the invention of the program-controlled computer as we know it and is paradoxically marked by a negative result. In 1936, Alan M. Turing introduced the notion of computing machines (later to be known as *Turing machines*) and showed that there is no general algorithm to decide whether a calculation performed by such a machine will eventually terminate and yield a result [Tur36]. An immediate corollary of this finding is that the validity of correctness assertions about a program cannot be checked automatically in general.

This observation did not curb the enthusiasm of the early software pioneers. Turing as well as his contemporaries recognised the importance of correctness arguments. Herman Goldstine and John von Neumann proposed the use of assertion boxes to "indicate the logical situation at selected points" in the flow diagram describing a program [GvN47]. Turing himself later published a paper on verifying the correctness of software routines in which he emphasised the importance of annotating programs with assertions [Tur49]. The full potential of assertions was unveiled by Robert W. Floyd, who used them to attach logical interpretations to programs [Flo67]. He argued that the rigorous annotation of programs with assertions enables one to prove "by induction on the number of commands executed" whether or not a certain assertion holds upon completion of the execution. This work was a direct precursor to Tony Hoare's system of axioms and inference rules for the verification of programs, which has attained considerable fame under the name of "Hoare Logic" [Hoa69]. Hoare Logic links assertions and programming constructs by means of Hoare triples. A Hoare triple comprises a precondition, a command, and a postcondition, and states that whenever the precondition is met a terminating execution of the command establishes the postcondition. Edsger W. Dijkstra developed this idea further and introduced the notion of predicate transformers such as the weakest precondition (later to be complemented by the strongest postcondition), which are total function mappings between preconditions and postconditions [Dij75].

Owing to Turing's undecidability result, neither Hoare logic nor Dijkstra's calculus en-

[1] `http://www.comlab.ox.ac.uk/publications/publication3240-abstract.html`

able fully mechanised program verification. In both systems, reasoning about iterative language constructs depends on the choice of appropriate invariant assertions, i.e., assertions that hold before and after each iteration of the repetitive construct. Given these inductive assertions, an automated program verifier (such as the one presented by James Cornelius King in his dissertation [Kin70]) can infer the intermediate assertions by means of symbolic simulation [HK76] and check the consistency of the resulting Hoare triples using a sufficiently powerful theorem prover [KF70]. Finding the invariants, however, often requires considerable insight.

Dijkstra's recursive definition of predicate transformers for repetitive commands (cf. [Dij75], Section 3.3), however, suggests that these invariants can be computed for at least certain instances of programs. Edmund Melson Clarke discovered that the completeness of a Floyd-Hoare axiom system depends on the existence of a fixed point for Dijkstra's predicate transformers [Cla77].

Clarke's observation of the relation between program invariants and fixed points later led to a major breakthrough in the field of automated program verification: In 1981, Clarke and his student E. Allen Emerson presented "a *model checking* algorithm which can be applied to mechanically verify that a finite state concurrent program meets a particular Temporal Logic specification" [CE81]. A similar result was presented independently by Jean-Pierre Queille and Joseph Sifakis [QS82]. The algorithm exhaustively explores the state space of a program by iteratively growing the set of explored reachable states until a fixed point is reached. Model checking enables fully automated program verification (albeit only of finite state programs) and earned its inventors the prestigious Turing award (placing them in good company with Dijkstra, Floyd, and Hoare).

The next milestone in the history of automated verification was Kenneth L. McMillan's invention of symbolic model checking [BCM+90, McM93]. McMillan applied Randal E. Bryant's binary decision diagrams [Bry86] (BDDs) as an efficient data structure to symbolically represent finite sets of program states. This new approach enabled the verification of hardware designs with state spaces ten orders of magnitude larger than could be handled by means of explicit state enumeration at that time.

A central feature of model checking algorithms is their ability to report execution traces

that falsify assertions which do not hold. The direct correspondence of such *counterexamples* to failed test cases makes their value to engineers evident. The utility of counterexamples for software verification, however, goes far beyond falsification: their main additional merit lies in their applications in automated abstraction (a detailed survey can be found in [CV03]).

The combination of model checking and abstraction [CC77], the latter a technique which had long since been applied in the field of static analysis, finally enabled the verification of real world software programs. The advent of predicate abstraction [GS97] brought the worlds of model checking and Hoare logic together. Predicate abstraction made it possible to mechanise the creation of approximate finite state models of software by constructing Hoare triples over a fixed and finite set of assertions. The success or failure of this technique is still determined by the choice of an appropriate set of assertions. A poorly chosen set of assertions which is insufficient to prove the property in question results in *spurious* counterexamples.

Counterexample-guided abstraction refinement [BSV93, Kur94, CGJ^{+}00] (CEGAR) is based on the observation that the information provided by a spurious counterexample can be used to improve the accuracy of the abstract model. In the context of predicate abstraction, this is achieved by extracting assertions from counterexamples which are sufficient to eliminate the corresponding execution trace from the current abstraction [BR02a]. In the CEGAR approach, the approximate model is repeatedly refined until the property in question can be either falsified by a genuine counterexample or verified using a sufficiently accurate set of assertions. A number of successful automated software verification tools implement this approach (e.g., [BCLR04, HJMS02, CKSY05]).

At the same time, the significant advances in the field of automated decision procedures for propositional logic (and SAT solvers in particular) raised the incentive to replace BDDs with efficient Boolean decision procedures. This led to symbolic bounded model checking [BCCZ99] (BMC), an algorithm which inspects execution traces up to a bounded length only. This technique generates counterexamples much faster than BDD-based algorithms, but, due to its limited scope, typically fails to conclusively show the correctness of programs. Accordingly, its focus lies on falsification rather than on verification.

The success of BMC pinnacled in software verification tools that do away with ab-

straction entirely and inspect programs by symbolically simulating all execution traces of a program up to a certain, user-defined length (e.g., [CKSY04, IYG+08]). Despite its obvious limitations and its inability to reveal bugs that are deeply hidden in a program, this technique is widely (and successfully) applied in industry.

An inherent deficiency (or rather, intentional design decision) of BMC is the lack of an adequate mechanism to detect when a program has been exhaustively explored. The proposed estimates for a sufficient bound are either not tight enough or computationally expensive to derive from the model. In particular, they are typically not applicable to software. While k-induction [SSS00], an approach to compute invariants by means of induction has been successfully applied for hardware models, the first attempts to apply this technique to software are only very recent [DKR10].

McMillan presented a promising approach to derive invariants by means of Craig interpolation [McM03]. Interpolant-based model checking is an approximate method for computing invariants of symbolic transition systems. Intuitively, interpolants over-approximate the safe states reached by program executions that do not violate the property being checked. An interpolant can be efficiently extracted from a proof of safety (generated by a decision procedure) for a finite length execution trace. While the technique has its origins in hardware model checking, it was soon realised that it can also be applied in a predicate-abstraction based CEGAR framework to derive refinement predicates from spurious counterexamples [HJMM04]. These predicates were found to be more "sparse" and concise than the assertions computed using traditional techniques such as the weakest precondition. A conceptually similar approach, which also uses interpolation to derive assertions but does not rely on predicate abstraction was presented by McMillan [McM06].

The utility of Craig interpolants in verification was noticed only recently, though the notion of interpolation was introduced by William Craig over half a century ago [Cra57a]. Initially, the generation of Craig interpolants was based on specialised, proof-generating, decision procedures (such as [McM05]). This role is increasingly taken over by Satisfiability Modulo Theory (SMT) decision procedures [NOT06], which appear to be superseding SAT solvers and BDDs as the driving force powering modern model checking tools. Interpolation-based model checking and interpolating decision procedures are an evolving field, and many

issues are still not well understood. So far, interpolation seems to be a promising direction to take us one step further along the path towards finding better invariants.

1.2 A Mission Statement

Despite the numerous advances outlined in the previous section, the following facts have invariably remained unchanged throughout the history of software verification:

- Programmers only provide incomplete specifications for their code. Even modern programming languages still rely on assertions as the primary (and often only) means of specifying correct behaviour.
- The pivotal problem of automating the verification of these assertions is to find appropriate invariants.
- If an assertion does not hold, counterexamples (or failed test cases) are indispensable when it comes to understanding why this is the case.

Academic programming languages such as Eiffel [Mey92] address the first two issues by allowing software engineers to annotate their programs with invariants. In practice, however, this methodology has not yet reached wide acceptance (partially due to the lack of automated verification tools for these annotations). Modal formalisms, such as temporal logic (the lingua franca of hardware verification engineers) have no equivalent in the world of software engineering.

The kind of properties that can be specified using assertions are known as safety and reachability properties. Since this dissertation does not have the ambition to change the established practices of an entire industry, its focus is on the verification of safety properties only. It should be noted, though, that for finite state models the verification of liveness properties can be reduced to safety checking [BAS02]. Accordingly, tools that check program termination (see, for instance [CPR05]) can benefit directly from the contributions of this dissertation.

Testing is currently the predominant verification technique for software. It owes its popularity to its scalability and the ability to produce counterexamples. The latter feature

is also provided by software model checking tools. However, the presence of loops in the program under test may have a strong negative effect on the performance of CEGAR-based model checking tools [KW06]. Part of this dissertation is therefore dedicated to accelerating the detection of counterexamples.

Counterexamples are a result of failed verification attempts [Pop34]. In the case of model checking, verification and falsification are two sides of the same coin and are strongly intertwined. It is therefore appropriate to address both aspects in one dissertation. The aim of this dissertation is to improve over the state of the art in the field of CEGAR-based software model checking for verification as well as falsification. In particular, the main agenda items are:

- Generating inductive invariants by means of Craig interpolation. We present a technique that is able to provide a range of different interpolants and allows us to fine-tune their logical strength.
- Accelerating the detection of counterexamples. We present an approach that, for a certain class of programs, efficiently finds counterexamples which contain a large number of loop iterations.

Before we detail our contributions in Section 1.4, we present motivating examples for the points listed above.

1.3 Two Motivating Examples

This section contains two exemplary verification problems illustrating the challenges that arise in automated verification. For the sake of understandability we keep the examples small and simple. We emphasise, however, that they are motivated by realistic scenarios that may occur in real programs – bit-vector operations and buffer overflows.

The examples we present are small enough to be amenable to paper-and-pencil reasoning. For larger examples, however, manual verification is arguably neither feasible nor desirable. Therefore, we present and apply machinery to mechanise this tedious task (in particular, the model checking tools Wolverine and SatAbs [CKSY05]) in this dissertation.

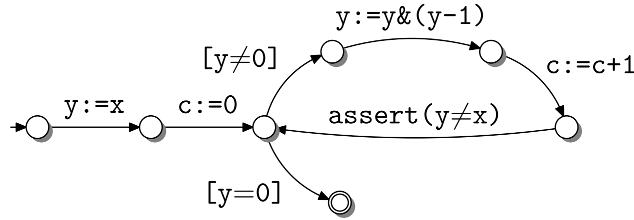


Figure 1.1: Wegner's algorithm to count ones [Weg60].

We present one correct program and one program which does not satisfy its specification, establishing a balance between verification and falsification which we aim to maintain throughout this dissertation. Both concepts are, in our opinion, equally important.

We use control-flow-automata (CFA) [HJM+02] to represent programs (see, for instance, Figure 1.1). The transitions (or edges, respectively) of a CFA are annotated with assignment statements, conditions, and assertions. The latter constitute the specification the program is supposed to satisfy. To avoid confusion with the assertions in Hoare triples, we refer to Hoare's pre- and postconditions as predicates from now on. We defer a formal definition of CFA and their semantics to Chapter 2.

The goal of the verification process is to annotate each node in the CFA with a predicate such that each annotation of an edge forms a Hoare triple with the predicates of the adjacent nodes. If such an annotation does not exist, we desire a justification for the failure of the verification process in form of a counterexample.

1.3.1 Verification

The program in Figure 1.1 computes the number of bits set to one in the binary representation of variable `x`. This is achieved by clearing the least significant bit set in each iteration of the loop [Weg60]. While the assertion `x≠y` represents an invariant at the respective program location, the invariant is not inductive, i.e., the condition `x≠y` alone is not sufficient to guarantee that the assertion still holds after an additional iteration of the loop. In the following, we attempt three different techniques to derive an appropriate inductive invariant: Computing the strongest postcondition and weakest precondition of a path, and Craig interpolation.

Table 1.1: Predicate transformers for statements in control-flow-automata

Statement `stmt`	**Strongest Postcondition** $sp(\texttt{stmt}, P)$	**Weakest Precondition** $wp(\texttt{stmt}, Q)$
$\texttt{x:=}e$	$\exists \texttt{x}' . (\texttt{x} = e[\texttt{x}/\texttt{x}']) \wedge P[\texttt{x}/\texttt{x}']$	$Q[\texttt{x}/e]$
$[R]$	$P \wedge R$	$R \Rightarrow Q$
$\texttt{assert}(R)$	$P \wedge R$	$Q \wedge R$
$\texttt{stmt}_1 ; \texttt{stmt}_2$	$sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P))$	$wp(\texttt{stmt}_1, wp(\texttt{stmt}_2, Q))$

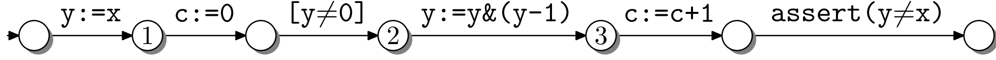


Figure 1.2: A path in the control flow graph in Figure 1.1

Predicate Transformers

Dijkstra's predicate transformers ([Dij75], cf. Section 1.1) are, at least for the restricted set of program statements we allow as annotations of CFA edges, purely syntactical operations on predicates. Table 1.1 introduces the strongest postcondition and weakest precondition for assignment statements, conditions, and assertions. We use $\texttt{x}$ and $\texttt{x}'$ to denote variables and e to denote an expression in a not yet further specified first-order language. The term $e[\texttt{x}/\texttt{x}']$ denotes the expression e with all free occurrences of $\texttt{x}$ replaced by $\texttt{x}'$.

Hantler and King [HK76] states that "if a program contains no branches, symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit". In particular, the predicate transformers in Table 1.1 are sufficient to perform a symbolic execution of CFA-paths of finite length.

The strongest postcondition enables us to compute a symbolic representation of the program states reachable at each location of the path. We refer to this technique as forward simulation. For a path which ends in an assertion which it does not violate, the weakest precondition allows us to determine the set of safe states for each location of the path. This approach is known as backward simulation. In both cases, the simulation starts from the predicate true, indicating that there are no constraints on the initial program states or the states after the assertion.

Figure 1.2 shows a path of the program in Figure 1.1. We label three of the nodes with ordinals. Table 1.2 lists the respective strongest postconditions and weakest preconditions for each of these nodes. For convenience and readability, we have eliminated the existential

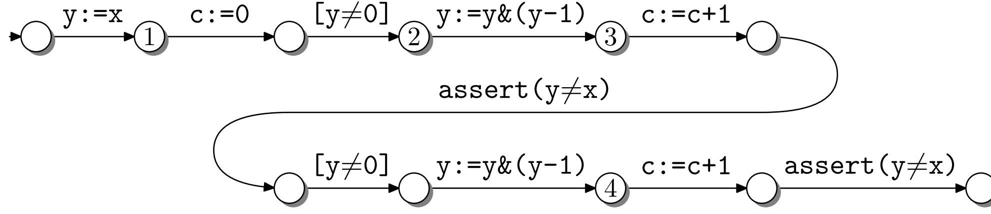


Figure 1.3: A path in the control flow graph in Figure 1.1.

quantifiers introduced by the *sp*-transformations. Furthermore, we ignore statements updating the counter `c`, since they do not affect the correctness of the program (this approach is known as slicing [Wei81]). By construction, the predicates listed in Table 1.2 are sufficient to prove that the path in Figure 1.2 does not violate the assertion.

The predicates derived using *sp* and *wp*, however, are not inductive. The predicate $\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0$ is too strong to be useful for proving the correctness of the program in Figure 1.1. To see this, consider the path in Figure 1.3. This path iterates the loop in Figure 1.1 a second time. We use the strongest postcondition of the predicate at location ③ to construct a predicate for node ④:

$$sp([\mathtt{y} \neq \mathtt{0}],\ (\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0)) \;=\; (\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0) \wedge (y \neq 0)\,, \tag{1.1}$$

$$\begin{gathered} sp(\mathtt{y} := \mathtt{y}\&(\mathtt{y}-1),\ (\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0) \wedge (y \neq 0)) = \\ (\mathtt{x}\&(\mathtt{x}-1) \neq 0) \wedge (\mathtt{x} \neq 0) \wedge \mathtt{y} = (\mathtt{x}\&(\mathtt{x}-1))\&((\mathtt{x}\&(\mathtt{x}-1)) - 1)\,. \end{gathered} \tag{1.2}$$

The assertion is redundant and can therefore be ignored. The predicate derived in (1.2) does not imply the predicate at location ③. In fact, it contradicts $\mathtt{y} = (\mathtt{x}\&(\mathtt{x}-\mathtt{1}))$.

Conversely, it can be shown that the predicate $x \neq y$ derived for location ③ using *wp* is too weak to be an inductive invariant. In order to obtain an inductive invariant by means of the predicate transformers discussed in this section, we need to iteratively unwind the loop in the example program until the disjunction of all predicates for the node following the statement `x:=y&(y-1)` reaches a fixed point. If the bit-width of the variables `x` and `y` is n, this happens after n iterations.

Step	Strongest Postcondition	Interpolant	Weakest Precondition
①	$\mathtt{x} = \mathtt{y}$	$\mathtt{y} \leq \mathtt{x}$	$(\mathtt{x} \neq \mathtt{y}\&(\mathtt{y}-1)) \vee (\mathtt{y} = 0)$
②	$\mathtt{x} = \mathtt{y} \wedge \mathtt{y} \neq 0$	$\mathtt{y} \leq \mathtt{x} \wedge \mathtt{y} \neq 0$	$(\mathtt{x} \neq \mathtt{y}\&(\mathtt{y}-1))$
③	$\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0$	$\mathtt{x} \neq 0 \wedge \mathtt{y} \leq (\mathtt{x}-1)$	$\mathtt{x} \neq \mathtt{y}$

Table 1.2: Strongest postconditions, interpolants, and weakest preconditions for the path in Figure 1.2

Craig Interpolation

As the names of the predicate transformers discussed in the previous section indicate, the predicates derived by means of *sp* and *wp* are the most extremal choices possible. Typically, there is a whole range of alternatives above and below (in the implication order) the predicates generated using *sp* and *wp*, respectively. The art of software verification boils down to choosing appropriate candidates from this range.

In the following, we describe Craig interpolants [Cra57a], adapted to our setting, in an intuitive manner. We formalise and generalise this notion in Chapters 2 and 3. Let ① be a node in a CFA-path of finite length ending in an assertion which it does not violate. A Craig interpolant for ① is a predicate I such that

- I is implied by the predicate for ① generated using the strongest postcondition.
- I implies the predicate for ① generated using the weakest precondition.
- I refers only to symbols which are in the syntactical scope at the node ①.

In our setting, we require furthermore that interpolants for the nodes of a path are inductive in the sense that

- if I and J are interpolants for the nodes preceding and following (respectively) an edge of the path annotated with a statement `stmt`, then $sp(\mathtt{stmt}, I)$ implies J.

The middle column of Table 1.2 shows such a set of interpolants for the labelled nodes of the path in Figure 1.2. Given an adequate set of logical inference rules (such as $(x = y\&z) \vdash (x \leq y) \wedge (x \leq z)$), it is possible to verify that the conditions above hold.

For a number of logical languages commonly used in program verification, it is possible to derive interpolants from a proof of safety for a path (cf., for instance, [McM05]). By "proof

$$\dfrac{\dfrac{\dfrac{(\mathtt{y}'{=}\mathtt{x}) \qquad \mathtt{y}' \neq 0}{(\mathtt{x}{\neq}\ 0)} \qquad \dfrac{\exists \mathtt{y}'\,.\quad \mathtt{y} = \mathtt{y}'\&(\mathtt{y}'-1) \quad \wedge \quad \mathtt{y}' = \mathtt{x}}{\mathtt{y} = \mathtt{x}\&(\mathtt{x}-1)}}{\mathtt{x} \neq 0 \ \wedge \ (\mathtt{y} \leq (\mathtt{x}-1))}}{\mathtt{y} \neq \mathtt{x}}$$

Figure 1.4: A proof of safety for the path in Figure 1.2

of safety for a path" we denote a rule-based derivation establishing that the assertion at the end of the path cannot be violated. For this purpose, we use the *sp* predicate transformer to construct a formula corresponding to the prefix of the path up to the assertion, and show that this formula implies the assertion.

Figure 1.4 shows such a proof of safety for the path presented in Figure 1.2. (A discussion of the inference rules used in this proof is provided in Section 3.5.) The antecedents of the proof derive immediately from the strongest postconditions at the nodes ①, ②, and ③ of the path. The intermediate results in the proof are obtained by means of logical inference rules of the kind mentioned above. The final step in the proof establishes that the assertion $\mathtt{y}{\neq}\mathtt{x}$ must hold.

Note that all subformulae of the interpolants listed in Table 1.2 occur in the proof in the antecedents or inferred predicates. Kovács and Voronkov [KV09b] shows that this is no coincidence. Given a proof that has a certain structure such as the one in Figure 1.4, there is always an interpolant which is a Boolean combination of the conclusions of the inferences (cf. [KV09b], Lemma 10).

We proceed to show that the interpolant $\mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1))$ for node ③ is an inductive invariant for the loop in Figure 1.1. Recall the path in Figure 1.3. We compute accordingly

$$sp([\mathtt{y} \neq \mathtt{0}],\ \mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1))) \ = \ \mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1)) \wedge (\mathtt{y} \neq 0)\,, \tag{1.3}$$

$$\begin{aligned} sp(\mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1}),\ \mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1)) \wedge (\mathtt{y} \neq 0) \ =& \\ \exists \mathtt{y}'\,.\, \mathtt{x} \neq 0 \wedge (\mathtt{y}' \leq (\mathtt{x}-1)) \wedge (\mathtt{y}' \neq 0) \wedge (\mathtt{y} = \mathtt{y}'\&(\mathtt{y}'-1))\,.& \end{aligned} \tag{1.4}$$

From (1.4) we can derive by means of inference rules such as $(x = y\&z) \vdash (x \leq y)$ and transitivity that $\mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1))$ must hold at node ④. The proof is outlined in Figure 1.5. The predicate $\mathtt{x} \neq 0 \wedge (\mathtt{y} \leq (\mathtt{x}-1))$ is an inductive invariant. It is also

$$\dfrac{\dfrac{\dfrac{\exists \mathtt{y}' \,.\, \mathtt{x} \neq 0 \wedge (\mathtt{y}' \leq (\mathtt{x}-1)) \wedge (\mathtt{y}' \neq 0) \wedge (\mathtt{y} = \mathtt{y}'\&(\mathtt{y}'-1))}{\mathtt{x} \neq 0 \quad \wedge \quad \exists \mathtt{y}' \,.\, (\mathtt{y}' \leq (\mathtt{x}-1)) \wedge (\mathtt{y} = \mathtt{y}'\&(\mathtt{y}'-1))}}{\mathtt{x} \neq 0 \quad \wedge \quad \exists \mathtt{y}' \,.\, (\mathtt{y}' \leq (\mathtt{x}-1)) \wedge (\mathtt{y} \leq \mathtt{y}')}\;\boxed{(x = y\&z) \vdash (x \leq y)}}{\boxed{\mathtt{x} \neq 0 \;\wedge\; (\mathtt{y} \leq (\mathtt{x}-1))}}\;\boxed{x \leq y \wedge y \leq z \vdash x \leq z}$$

Figure 1.5: Proof of inductivity for the invariant $\mathtt{x} \neq 0 \;\wedge\; (\mathtt{y} \leq (\mathtt{x}-1))$

an interpolant for node ④, since the proof in Figure 1.5 can easily be extended to show that $\mathtt{x} \neq \mathtt{y}$ (cf. Figure 1.4). Since this new interpolant implies a previously encountered interpolant for the same program location, we can conclude that we have reached a fixed point and that the program is safe.

This completes the argument that the interpolants in Table 1.2 can be used to construct valid Hoare triples for all edges in the program in Figure 1.1, proving the assertion $\mathtt{y}{\neq}\mathtt{x}$ correct. Before we proceed to the next example, we note that the proofs in Figures 1.4 and 1.5 make repeated use of inference rules from the theory of bit-vectors. This is crucial to guarantee the soundness of the verification process. Computers store values of variables using a binary representation. The semantics of the operations on these binary values is determined by their gate-level implementation in the central processing unit (as indicated for the decrement in Figure 1.6). This means that the range of variables is limited and that an arithmetic operation can result in an overflow. In particular, the predicate $(\mathtt{y} \leq (\mathtt{x}-1))$ does not imply that $\mathtt{x}{\neq}\mathtt{y}$: Let $\mathtt{x}$ and $\mathtt{y}$ be unsigned 2-bit variables represented by $\langle x_1 x_0 \rangle$ and $\langle y_1 y_0 \rangle$, respectively. Consider that $\mathtt{x}$ is zero, i.e., $\langle x_1 x_0 \rangle = \langle 00 \rangle$. For this input, the result of the decrement operation in Figure 1.6 is $\langle s_1 s_0 \rangle = \langle 11 \rangle$, i.e., three. Therefore, $\mathtt{x}{=}0$ and $\mathtt{x}{=}\mathtt{y}$ satisfies $(\mathtt{y} \leq (\mathtt{x}-1))$, and a decision procedure that concludes otherwise may possibly result in an invalid correctness argument for a program. In such a case, the verification tool must not conclude that the program is correct.

Many efficient decision procedures for bit-vector logic are (at least partially) based on a propositional encoding of the formula. This approach is known as bit-blasting and enables the deployment of efficient, off-the-shelf SAT solvers. While it is possible to extract Craig interpolants from the propositional resolution proofs generated by these solvers, the resulting interpolants are propositional formulae rather than instances of the word-level

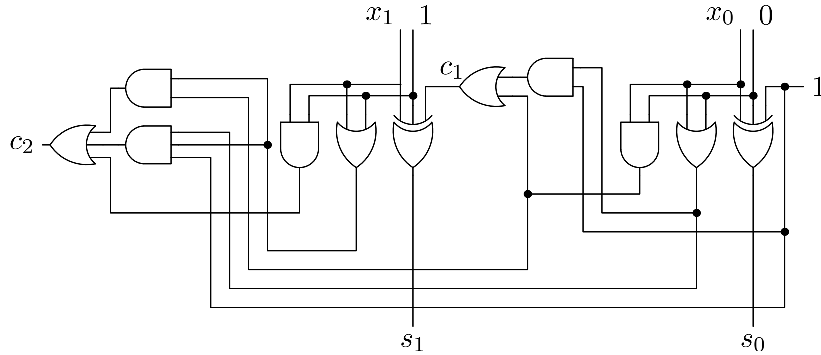


Figure 1.6: The semantics of the 2-bit bit-vector decrement-by-one operation [BKWW08].

language in which the original problem was formulated. The propositional encoding of a formula such as $(\mathtt{y} \leq (\mathtt{x}-1))$ is based on the gate-level implementation of the binary relation $\leq$ and the decrement operation (Figure 1.6). Such a bit-level encoding is unwieldy and hard to interpret for a human. Accordingly, it is preferable to avoid such predicates in Hoare proofs. Chapter 3 presents a technique that allows us to achieve this goal by combining bit-level and word-level reasoning.

1.3.2 Falsification

The mere information that the verification tool failed to prove an assertion safe is of little to no use to the programmer. The failure to prove the correctness of the program under test does not necessarily indicate the presence of a bug. If a bug is present, the existence of an actual violation of the assertion needs to be demonstrated by means of a counterexample.

Finding a counterexample is not a trivial task. A programming fault may only manifest itself under very particular conditions. The main challenge for verification tools based on symbolic simulation is that bugs are not necessarily "shallow". For instance, an execution trace triggering a buffer overflow may require a several hundred loop iterations. In a CEGAR-framework, this may result in repeated, computationally expensive refinement attempts until the bug is eventually unveiled.

The structure of the CFA in Figure 1.7 is that of a typical program containing a buffer overflow. The assertion replaces the access to an array or a buffer large enough to hold 100 elements and checks whether the index `j` exceeds the upper bound. The search for an inductive invariant establishing that the assertion $\mathtt{j} < 100$ holds must necessarily fail, since

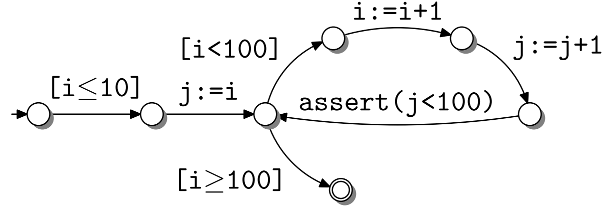


Figure 1.7: A program with a loop and an upper bound on j

such an invariant does not exist.

The iterative verification approach outlined in Section 1.3.1 effectively performs a symbolic forward simulation in this case. If the program contains branches, this technique results in a reachability tree [HK76], which contains all unwindings of the loop up to the point where the violation of the assertion is found. Figure 1.8 shows the shortest path which possibly violates the assertion. The repeat symbols indicate that the enclosed sequence of statements should be replaced by a concatenation of 89 copies of itself. If we compute the strongest postconditions for the nodes ①, ②, and ③ we obtain:

$$\begin{aligned} sp([\mathtt{i} \le 10]; \mathtt{j} := \mathtt{i},\ \mathsf{true}) &= & (\mathtt{i} \le 10) \wedge (\mathtt{j} = \mathtt{i}) \quad ① \\ sp([i < 100]; \mathtt{i} := \mathtt{i} + 1,\ (\mathtt{i} \le 10) \wedge (\mathtt{j} = \mathtt{i})) &= & (\mathtt{j} \le 10) \wedge (\mathtt{i} \le 11) \wedge (\mathtt{i} < 101) \quad ② \\ sp(\mathtt{j} := \mathtt{j} + 1,\ (\mathtt{j} \le 10) \wedge (\mathtt{i} \le 11) \wedge (\mathtt{i} < 101)) &= & (\mathtt{j} \le 11) \wedge (\mathtt{i} \le 11) \wedge (\mathtt{i} < 101) \quad ③ \end{aligned}$$

Accordingly, the assertion $\mathtt{j} < 100$ following node ③ holds. Simulating an additional iteration of the loop up to node ④ yields

$$sp([\mathtt{i} < \mathtt{100}];\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{j} := \mathtt{j} + 1,\ (\mathtt{j} \le 11) \wedge (\mathtt{i} \le 11) \wedge (\mathtt{i} < 101)) = \\ (\mathtt{j} \le 12) \wedge (\mathtt{i} \le 12) \wedge (\mathtt{i} < 101)\,,$$

still short of violating the assertion. (Note that we also did not reach a fixed point.) Only after 88 additional iterations of the loop we obtain a strongest postcondition of $(\mathtt{i} \le 100) \wedge (\mathtt{j} \le 100)$. Since this predicate does not rule out a violation of the assertion $(\mathtt{j} < 100)$, we are finally in a position to report a counterexample.

The reason for the repeated and wasteful iteration is that the verification technique

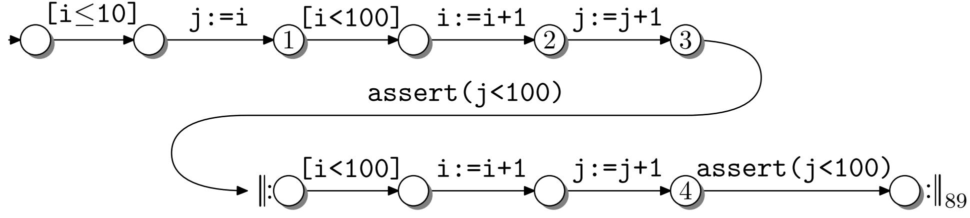


Figure 1.8: A long counterexample violating the upper bound

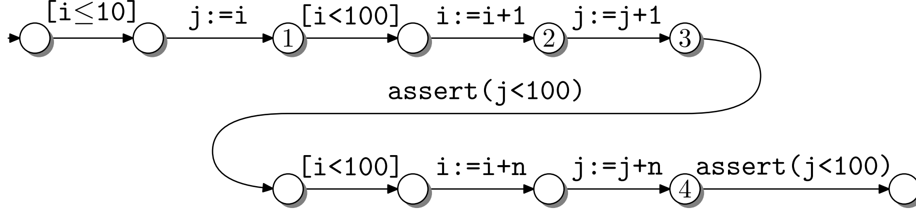


Figure 1.9: A parametrised counterexample violating the upper bound

ignores the structure of the program. In Chapter 4 we propose a technique to accelerate the detection of counterexamples. The underlying idea is to detect loops which, if iterated sufficiently often, potentially violate an assertion. We then parametrise the number of iterations in the strongest postcondition.

Consider the path in Figure 1.9. After simulating the first few steps of the path, we revisit the entry node of the loop in the program in Figure 1.7. Since we anticipate that the loop will be executed repeatedly, we replace the assignments `i:=i+1` and `j:=j+1` by a parametric version `i:=i+n` and `j:=j+n`. Computing the strongest postcondition for node ④ yields

$$sp([\mathtt{i} < \mathtt{100}];\, \mathtt{i} := \mathtt{i} + \mathtt{n};\, \mathtt{j} := \mathtt{j} + \mathtt{n},\ (\mathtt{j} \leq 11) \wedge (\mathtt{i} \leq 11) \wedge (\mathtt{i} < 101)) = \\ (\mathtt{j} \leq (11 + \mathtt{n})) \wedge (\mathtt{i} \leq (11 + \mathtt{n}))\,.$$

Using a constraint solver, we can now determine a value for $\mathtt{n}$ such that the strongest postcondition $(\mathtt{j} \leq 11 + \mathtt{n}) \wedge (\mathtt{i} \leq 11 + \mathtt{n})$ allows for a violation of the assertion $(\mathtt{j} < 100)$. It turns out that 89 is the smallest candidate for such an $\mathtt{n}$. This result represents an educated guess for the number of loop iterations required to violate the assertion. For our example, we can immediately determine that initialising `i` to 10 and iterating the loop 90 times does indeed trigger the assertion.

Finally, we want to stress that this technique is also able to accelerate the generation of inductive invariants. In case the constraint solver is unable to find a value for `n` which invalidates the assertion, we can extract an interpolant directly from a proof of safety for the parametrised path. We discuss this technique in further detail in Chapter 4 and demonstrate how it allows us to verify programs for which the approach outlined in Section 1.3.1 would fail.

1.4 Contributions

This dissertation presents contributions to the field of interpolation-based software model checking, an approach which has recently been the subject of significant research activity (see, for instance, [HJMM04, McM06, EKS06, HHP10]). The dissertation presents theoretical results as well as empirical results obtained by integrating some of the techniques we discuss into the automated verification tools WOLVERINE and SATABS.

Theoretical Results

We divide our theoretical contributions into techniques related to verification and falsification, respectively.

Verification. We present novel techniques to derive Craig interpolants from refutation proofs. Craig interpolants can be efficiently extracted from proofs that adhere to certain structural restrictions (see, for instance, [Mae61, Hua95, Pud97, Kra97, McM05, KV09b]). Many modern decision procedures (such as SAT and SMT solvers) are able to generate refutation proofs of this kind. We distinguish between "bit-level" decision procedures, which operate on a propositional encoding of the original problem instance and generate propositional resolution proofs, and "word-level" decision procedures, which treat variables as an entity and apply theory specific inference rules to derive a contradiction. The interpolation systems presented in this dissertation cover both approaches (as well as combinations thereof) and enable the generation of a wider range of interpolants than currently possible with existing techniques:

- We present a technique to fine-tune the logical strength of propositional interpolants derived from resolution proofs. Our technique generalises existing propositional interpolation systems (in particular [Hua95, Pud97, Kra97, McM05]) and enables the generation of a wider range of interpolants.

- We relate propositional reasoning and word-level decision procedures by showing how resolution proofs of a certain structure can be lifted to word-level proofs in the theory of bit-vectors. This contribution

  - exposes a structural relation of propositional resolution proofs and word-level proofs from which we are able to extract interpolants, and
  - enables the combination of bit-level and word-level reasoning for the purpose of generating word-level interpolants.

  The technique differs from traditional SMT techniques in the sense that it is an ex post facto approach, i.e., it is applied only after the SAT solver has reached its final verdict.

- The common practice of modelling the behaviour of bit-vector operations using arithmetic over the unbounded integers or reals introduces a discrepancy between the bit-level semantics of program variables and the domain of the resulting interpolants. Therefore, we propose an interpolating decision procedure which is sound for reasoning over the bit-vectors and can therefore be used in the context of the proof-lifting approach we present. It extends existing graph-based algorithms for deciding equality logic with uninterpreted functions with limited support for theory specific inference rules (as illustrated in Figures 1.4 and 1.5). Moreover, we present two instances of our interpolation technique allowing us to generate word-level interpolants of different logical strength.

Falsification. We present an approach to accelerate the detection of counterexamples containing a large number of iterations of loop constructs. The underlying idea is to detect potential loops in the reachability tree generated by means of symbolic simulation. This approach has two aspects:

- It attempts an educated guess regarding the number of iterations required to violate an assertion. The technique may thus avoid computationally expensive refinement attempts (including the repeated computation of interpolants using the verification techniques discussed above).

- We show that, if the attempt to find a counterexample fails, the detection of loops can also have a positive impact on the verification process. By taking the program structure into account when computing interpolants we are able to verify classes of programs for which an uninformed verification technique would fail.

Tools and Empirical Evaluation

We implemented and evaluated some of the techniques presented in the previous section in our software verification tools Wolverine and SatAbs [CKSY05]. The Wolverine tool is an implementation of the interpolation-based algorithm presented in [McM06] and relies on the interpolating word-level decision procedure presented in our dissertation. The Wolverine tool has been implemented in the scope of our doctoral work and is successfully applied to verify a benchmark set consisting of a number of Windows and Linux device drivers.

We emphasise that, while verification and falsification are strongly intertwined, the improvements achieved by means of the techniques presented in this thesis are orthogonal. Therefore, we evaluate our counterexample detection approach separately by making appropriate adjustments to the predicate-abstraction based model checking tool SatAbs. We apply the modified version of SatAbs to an open source mail delivery agent and the buffer overflow benchmark presented in [KHCL07]. We observe a performance improvement of up to 400% for some of these benchmarks.

The verification tools Wolverine and SatAbs are available from the websites

- `http://www.cprover.org/wolverine` and

- `http://www.cprover.org/satabs`,

respectively.

1.5 Significance of Contributions

The philosopher Thomas Samuel Kuhn distinguishes between paradigm-shifting revolutionary science and normal science [Kuh62]. He describes the latter as "puzzle solving", an attempt to enlarge the paradigm of a field and to corroborate the confidence in its underlying techniques. This dissertation is a specimen of the latter class.

Our work on interpolating decision procedures generalises several existing approaches. Existing techniques generate exactly one interpolant per refutation proof and partition. Our techniques lift this restriction and enable the generation of a whole range of interpolants. Thus, we can provide the verification tool with a choice of candidates for invariants. While this increases the potential that an appropriate inductive invariant can actually be generated by means of interpolation, we have not yet investigated heuristics to identify promising candidates.

The falsification technique presented in our thesis has the potential to increase the performance of model checking tools by several orders of magnitude without adding a significant overhead to the verification process.

This dissertation is based on a number of previously published and peer-reviewed articles. It restates and extends the results presented in these publications. It presents a number of novel contributions to the field of automated software verification, which have been acknowledged by expert reviewers in the verification community. We provide a detailed survey of this field in [DKW08]. The discussion of the state of the art in this field (Section 2 of this dissertation) draws extensively on this publication.

In the following, we map the contributions listed in Section 1.4 to the respective publications:

Verification.

- The technique to generate a range of propositional interpolants of varying strength is presented in [DPWK10]. The complete proofs of the theorems stated in this publication are provided in an extended technical report [DKPW09].

- The relation between propositional resolution proofs and proofs based on theory-

specific inference rules is covered in our publication on lifting propositional proofs to the word-level [KW07].

- In [KW09a], we present an interpolating decision procedure which supports equality logic, uninterpreted functions and transitive relations and provides ad-hoc support for axioms of the bit-vector theory.

Falsification. Our approach to accelerate falsification by detecting potential loops in counterexamples is presented in [KW06]. Its benefits for verification are discussed in [KW10].

1.6 Organisation of this Dissertation

Chapter 2 provides an introduction to predicate abstraction and interpolation-based model checking. First, it introduces the basic concepts underlying these verification techniques: It briefly discusses Hoare logic and uses this concept to define the semantics of control flow automata (CFA). It then carries on to explain how Hoare logic can also be used to define abstractions of CFA. Then, we describe how abstractions can be automatically constructed using predicate abstraction or Craig interpolation. We present two state-of-the-art model checking techniques based on counterexample-guided abstraction refinement, predicate abstraction and interpolation, and highlight the similarities of these approaches.

The structure of the remaining part of this dissertation is dictated by the outline of our contributions presented in Section 1.4.

Chapter 3 covers the generation of Craig interpolants. It fixes a quantifier-free fragment of first order logic with bit-vector operations (Section 3.1), which serves as our assertion language for Hoare triples. Furthermore, it provides a general introduction to the mechanisms underlying our interpolation techniques (Section 3.2).

The remaining chapter is subdivided into three sections, each corresponding to one of our main contributions in this category. Section 3.3 briefly discusses propositional encodings of bit-vector logic. It introduces resolution proofs and presents an interpolation system for resolution refutations enabling the generation of a range of interpolants of different logical strength. Furthermore, we discuss how it is possibly to vary the logical strength of the

resulting interpolants by modifying the refutation proofs. Section 3.4 relates propositional resolution proofs and theory specific word-level refutation proofs. We explain how to lift propositional proofs to proofs over the bit-vector language defined at the beginning of the chapter. In Section 3.5, we demonstrate how interpolants can be constructed from word-level refutation proofs. We present two different interpolation systems for word-level proofs which allow us to generate interpolants of different strength. Section 3.6 is dedicated to the experimental evaluation of our interpolation techniques.

Chapter 4 covers falsification and the generation of counterexamples. In Section 4.1, we introduce an algorithm to detect potential loops in counterexamples. Then we discuss how to derive from parametrised paths an educated guess for the number of iterations required to violate an assertion. We continue with an exposition of how information about the structure of counterexamples can be exploited for the purpose of finding inductive invariants (Section 4.2). Section 4.3 provides a panopticon of example programs which can be verified more efficiently using this approach. In Section 4.4 we provide a relative completeness argument for our approach. Section 4.5 is dedicated to the experimental evaluation of our implementation.

Finally, we provide some concluding remarks and a discussion of open issues and potential future work.

Appendix A discusses implementation-specific issues of the Wolverine verification tool and describes the implementation of our decision procedure for bit-vector arithmetic. Appendix B contains a number of proofs for theorems presented in Chapters 2 and 3.

Chapter 2

Interpolation-based Verification Techniques

This chapter starts with a summary of the preliminary concepts used throughout this dissertation. It revisits Hoare logic and Dijkstra's predicate transformers (based on [Hoa69, Dij75] and Greg Nelson's excellent exposition of Dijkstra's calculus [Nel89]). These systems for formal reasoning over programs have a twofold purpose in our setting:

- They are used to define the semantics of control flow automata (CFA) [HJM+02] and abstractions thereof.
- They serve as the primary tool to generate and to refine abstractions for the aim of constructing safety proofs for programs.

We introduce Craig interpolants [Cra57a] and discuss how interpolation can be used to define alternative predicate transformers. We show that interpolants satisfy the requirements for the predicates of Hoare triples.

We use predicate transformers to define the semantics of programs and introduce the concept of reachability trees, which can be obtained using symbolic simulation [HK76].

For programs of realistic size, the construction of reachability trees is made feasible by analysing an abstraction of the CFA. (This approach is an instance of the abstract interpretation framework [CC79].) We discuss predicate abstraction [GS97] as one means

to obtain such abstractions and define this concept in terms of Hoare logic (cf. [Bal05], where predicates transformers are used). Furthermore, we show that interpolants represent a sound approximation of safely reachable states. This property makes interpolation a useful technique for refining abstractions in a counterexample-guided abstraction refinement framework.

We proceed to integrate the techniques presented in this section into a verification framework: we present two state-of-the-art interpolation-based model checking techniques, which constitute instances of the framework outlined in this section, namely the predicate abstraction-based CEGAR framework (incarnations of which are presented in [BR02b, HJMS02] and [CKSY05]) and lazy abstraction with interpolants [McM06]. These techniques enable us to construct safety proofs for programs.

We conclude by providing a brief overview of existing verification tools and related work (Section 2.6). For detailed comparison of these tools, we refer the reader to Section 3 of [DKW08], the author's main contribution to this survey of automated software verification.

Contribution. This chapter presents existing interpolation-based techniques using a unified formalism.[1] Furthermore, it provides a self-contained tutorial for software model checking with interpolants.

2.1 Hoare Logic and Dijkstra's Predicate Transformers

Hoare logic is based on predicates representing assertions about program states. Let $\mathcal{C}$ be a concrete domain of program states. A set of program states $\mathcal{S} \subseteq \mathcal{C}$ is characterised using a predicate P which maps a state s to true (denoted by $P(s) = \mathsf{true}$) if and only if $s \in \mathcal{S}$. We assume these predicates to be instances of a first-order language $\mathcal{L}$ and defer a formal definition of this language to Section 3. In our notation, $\mathtt{x}$ is a (program) variable, e a quantifier-free expression (or term, respectively), and P, Q, R are Boolean predicates, all well-formed members of $\mathcal{L}$. In our setting, predicates are typically either quantifier-free or amenable to quantifier elimination. We consider only expressions and predicates that have a well-defined meaning in the context of the program.

[1] Based on our publication on predicate-abstraction based verification of programs with loops [KW10].

$$\frac{}{\{P[\mathtt{x}/e]\}\ \mathtt{x:=}e\ \{P\}}\ \text{assignment} \qquad \frac{}{\{P \Rightarrow Q\}\ \mathtt{[}\,P\,\mathtt{]}\ \{Q\}}\ \text{condition}$$

$$\frac{}{\{P \Rightarrow Q\}\ \mathtt{assert}\,(P)\ \{Q\}}\ \text{assertion} \qquad \frac{\{P\}\ \pi_1\ \{Q\} \quad \{Q\}\ \pi_2\ \{R\}}{\{P\}\ \pi_1\mathtt{;}\pi_2\ \{R\}}\ \text{composition}$$

$$\frac{P \Rightarrow Q \quad \{Q\}\ \pi\ \{R\} \quad R \Rightarrow S}{\{P\}\ \pi\ \{S\}}\ \text{consequence}$$

Figure 2.1: A selection of Hoare logic rules for simple programming language constructs

Definition 2.1.1 (Hoare Triple)**.** *A Hoare triple comprises a precondition, a statement, and a postcondition. The Hoare Triple* $\{P\}$ `stmt` $\{Q\}$ *means that if the statement* `stmt` *is executed in a state in which* P *holds, then* Q *is true in any state in which* `stmt` *may halt [Nel89]. We refer to* P *as the* precondition *and to* Q *as the* postcondition *of the Hoare Triple.*

An alternative definition of $\{P\}$ `stmt` $\{Q\}$ (also given in [Nel89]) states that for all states s and s', if `stmt` is executed in s and yields s', then $P(s)$ implies $Q(s')$.

Hoare logic provides axioms and inferences rules for reasoning about imperative programming language constructs. In our setting, a program statement is either a *condition*, i.e., a total map from the concrete domain of program states $\mathcal{C}$ to the Boolean domain $\mathbb{B}$, or an *assignment* mapping $\mathcal{C}$ into $\mathcal{C}$ [CC79]. Furthermore, since we are concerned with the safety of programs, we assume that programs are annotated with assertions. We use $\mathtt{[}P\mathtt{]}$ to denote conditions, $\mathtt{x:=}e$ to denote assignments, and $\mathtt{assert(}P\mathtt{)}$ to represent assertions, where P is always an unquantified predicate. A path π is either the empty sequence ε, a program statement, or the concatenation $\pi_1\mathtt{;}\pi_2$ of two paths.

Figure 2.1 shows a set of axioms and inference rules for these constructs. We illustrate their application in the Examples 2.1.1 and 2.2.1 below.

Dijkstra's predicate transformers [Dij75] are an extension of Hoare logic. In Dijkstra's calculus, statements are total functions associating preconditions to postconditions. Compared to Hoare triples, Dijkstra imposes "tighter" conditions on preconditions and postconditions. The following definition of the weakest liberal precondition exposes the relation

Table 2.1: Predicate transformers for simple program statements

Statement `stmt`	**Strongest Postcondition** $sp(\texttt{stmt}, P)$	**Weakest (Liberal) Precondition** $w(l)p(\texttt{stmt}, Q)$
$\texttt{x:=}e$	$\exists \texttt{x}' \,.\, (\texttt{x} = e[\texttt{x}/\texttt{x}']) \wedge P[\texttt{x}/\texttt{x}']$	$Q[\texttt{x}/e]$
$\texttt{[}R\texttt{]}$	$P \wedge R$	$R \Rightarrow Q$
$\texttt{assert(}R\texttt{)}$	$P \wedge R$	$Q \wedge R \qquad (R \Rightarrow Q)$
$\texttt{stmt}_1\texttt{;}\,\texttt{stmt}_2$	$sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P))$	$w(l)p(\texttt{stmt}_1, w(l)p(\texttt{stmt}_2, Q))$

between Hoare logic and Dijkstra's calculus.

Definition 2.1.2 (Weakest Liberal Precondition)**.** *The weakest liberal precondition (denoted as* $wlp(\texttt{stmt}, Q)$*) for a statement* `stmt` *with respect to a postcondition* Q *is the weakest predicate* P *(in the implication order) such that* $\{P\}\ \texttt{stmt}\ \{Q\}$ *holds, i.e.,* $\{P'\}\ \texttt{stmt}\ \{Q\}$ *is equivalent to* $P' \Rightarrow wlp(\texttt{stmt}, Q)$*.*

Hoare logic proofs and the weakest liberal precondition only provide partial correctness arguments: the termination of program statements needs to be addressed separately. This is sufficient in our setting, since we only attempt to prove safety properties. Dijkstra's seminal paper [Dij75], however, introduces a predicate transformer which requires termination:

Definition 2.1.3 (Weakest Precondition)**.** *The weakest precondition* $wp(\texttt{stmt}, Q)$ *for a program statement* `stmt` *with respect to a postcondition* Q *is the weakest predicate* P *(in the implication order) such that* $\{P\}\ \texttt{stmt}\ \{Q\}$ *holds and* `stmt` *is guaranteed terminate.*

Thus, the weakest liberal precondition $wlp(\texttt{stmt}, Q)$ represents a relaxation of the weakest precondition $wp(\texttt{stmt}, Q)$. For unconditionally terminating statements such as conditional statements and assignments, however, the definitions of the weakest liberal precondition wlp and the weakest precondition wp coincide. In accordance to [Nel89], we indicate this correspondence by using the notation $w(l)p$ in Table 2.1, which lists the predicate transformers for these statements (cf. Table 1.1 in Section 1.3.1):

- The predicate transformers wp and wlp for the assignment statement state that what the precondition guarantees for e must also hold for the assigned variable `x` in the postcondition Q. Here, $Q[\texttt{x}/e]$ denotes the predicate Q with all free occurrences of `x` replaced by the expression e.

- The predicate transformer for a condition $[R]$ and an assertion $\texttt{assert}(R)$ warrants that R holds if the execution succeeds. The difference between these two statements is that, intuitively, a failing condition does not allow us to make any assumption on the outcome, whereas executing an assertion which does not hold results in "no proper (non-looping) outcomes", i.e., non-termination.[2] The fact that assertions are modelled using non-termination is the reason for the two entries in the respective cell of Table 2.1.

- Finally, predicate transformers can be applied to paths by means of functional composition, where the empty sequence of statements ε corresponds to the identity mapping.

Notably, the definition of wp for the conditional statement $[R]$ in Table 2.1 violates the so called "Law of the Excluded Miracle", a restriction introduced in Dijkstra's original exposition of predicate transformers (see [Dij75], property 1 in Section 3.1). The requirement that $wp(\texttt{stmt}, \textsf{false}) = \textsf{false}$ always has to hold effectively bars partial program statements such as $[R]$. Nelson [Nel89] takes strong opposition against this restriction:

> "I concluded that the Law of the Excluded Miracle is like bubble sort: It is unfortunate that is has a catchy name because the world would be better off forgetting it."

Partial statements are omnipresent in the work related to this dissertation (see, for instance, [BMMR01, HJMS02, CKSY05, McM06]), and their semantics is subtle and therefore sometimes misrepresented[3] ([HJMS02], Section 5.2.1 as well as [CC79], Section 3.2, for instance, state that $w(l)p([R], Q) = R \wedge Q$, which is stronger than $R \Rightarrow Q$). In this dissertation, we choose the more general predicate transformer $w(l)p$ presented in [Nel89] over its restrictive counterpart in [Dij75].

The weakest (liberal) precondition imposes a backwards direction of the program analysis. Its counterpart predicate transformer, the *strongest postcondition*, enables reasoning in the opposite direction.

[2]Accordingly, in $\texttt{assert}(P)$ corresponds to $(P \rightarrow \mathit{Skip}) \boxtimes \mathit{Loop}$ in Nelson's formalism [Nel89].
[3]We thank Joseph N. Ruskiewicz for bringing this issue to our attention.

Definition 2.1.4 (Strongest Postcondition). *The strongest postcondition* $sp(\texttt{stmt}, P)$ *for a statement* `stmt` *with respect to a precondition* P *is the strongest predicate* Q *(in the implication order) such that* $\{P\}$ `stmt` $\{Q\}$ *holds, i.e.,* $\{P\}$ `stmt` $\{Q'\}$ *is equivalent to* $sp(\texttt{stmt}, P) \Rightarrow Q'$.

The strongest postcondition for our set of simple statements is defined analogously to $w(l)p$ in Table 2.1. Some brief comments are in order:

- The existential quantification introduced by the strongest postcondition for assignment statements essentially takes the role of the substitution performed by $w(l)p$: What the precondition P states about `x` must hold of $\texttt{x}'$ (a fresh variable not occurring in P and e) in the postcondition.
- The strongest postcondition of $[R]$ simply warrants that R holds in addition to what the precondition already guarantees. If the condition $[R]$ does not hold for any of the states characterised by the precondition P, then the respective postcondition is **false**, which only holds for the empty set of states.
- The order in which the predicate transformer functions are composed is reversed.

Reasoning with the strongest postcondition corresponds to a *forward* reachability analysis (or symbolic simulation, respectively) of paths [HK76]. The strongest postcondition enables the construction of predicates characterising the set of reachable states at each point in the path. Each assignment statement $\texttt{x} := e$ gives rise to a fresh variable $\texttt{x}'$ representing the value of `x` before the execution of the statement. We briefly point out the similarity to two other common techniques in static analysis:

- Predicative programming [Heh84] treats statements as predicates over the initial and final value of `x` ($\grave{\texttt{x}}$ and $\acute{\texttt{x}}$, respectively). Dijkstra et al. [D+82] was the first to observe the correspondence of this approach and predicate transformers. The weakest liberal precondition for a statement `stmt` is completely determined by the predicate $wlp(\texttt{stmt}, \texttt{x} \neq \acute{\texttt{x}})[\texttt{x}/\grave{\texttt{x}}]$ (a proof is provided in [Nel89], Section 7). This predicate is the complement of the predicate representing `stmt` in predicative programming. Similarly, the predicate $sp(\texttt{stmt}, \grave{\texttt{x}} = \texttt{x})[\texttt{x}/\acute{\texttt{x}}]$ is a relation representing the statement `stmt`.

- The static single assignment form (SSA) [CFR+91] of a program is an intermediate representation used in compiler construction in which each variable is assigned exactly once. This is achieved by introducing subscripts for the original variables, e.g., $\mathtt{x}_1 := \mathtt{x}_0 + 1$ is a possible SSA representation of $\mathtt{x} := \mathtt{x} + 1$.

For paths, both representations can be obtained by means of *skolemisation* (see, for instance, [BJ89]) of the strongest postcondition and variable renaming. For instance, $\exists \mathtt{x}' . \mathtt{x} = \mathtt{x}' + 1$ is equi-satisfiable with $\mathtt{x}_1 = \mathtt{x}_0 + 1$ and $\acute{\mathtt{x}} = \grave{\mathtt{x}} + 1$ (which is equivalent to $\neg wlp(\mathtt{x} := \mathtt{x} + \mathtt{1},\, \mathtt{x} \neq \acute{\mathtt{x}})[\mathtt{x}/\grave{\mathtt{x}}]$). The representations are thus interchangeable in the context of the satisfiability of formulae. For convenience, we represent statements as predicates or in SSA form whenever it simplifies the presentation.

The direct correspondence of the rules in Figure 2.1 and the predicate transformers in Table 2.1 follows immediately from Definitions 2.1.2 and 2.1.3. The following example illustrates this tight connection.

Example 2.1.1. *We revisit the example presented in Section 1.3.1 of the Introduction. Consider the following slice of the path presented in Figure 1.2:*

$$\mathtt{y} := \mathtt{x};\ [\mathtt{y} \neq \mathtt{0}];\ \mathtt{y} := \mathtt{y}\&(\mathtt{y} - \mathtt{1});\ \mathtt{assert}(\mathtt{y} \neq \mathtt{x})$$

Table 1.2 shows the strongest postconditions and weakest preconditions for each point in this path. It is possible to obtain the same pre- and postconditions listed in this table by using the Hoare logic rules in Figure 2.1. In the following, we prove that the assertion at the end of this path cannot be violated. This can be achieved by showing that $\mathtt{y} \neq \mathtt{x}$ *holds immediately before the assertion. We obtain:*

$$\begin{array}{rr}
\{\mathtt{y}\&(\mathtt{y}-\mathtt{1}) \neq \mathtt{x}\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1})\ \{\mathtt{y} \neq \mathtt{x}\} & \textit{(assignment)} \\
\{\mathtt{y} = \mathtt{0} \vee \mathtt{y}\&(\mathtt{y}-\mathtt{1}) \neq \mathtt{x}\}\ [\mathtt{y} \neq \mathtt{0}]\ \{\mathtt{y}\&(\mathtt{y}-\mathtt{1}) \neq \mathtt{x}\} & \textit{(condition)} \\
\{\mathtt{x} = \mathtt{0} \vee \mathtt{x}\&(\mathtt{x}-\mathtt{1}) \neq \mathtt{x}\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} = \mathtt{0} \vee \mathtt{y}\&(\mathtt{y}-\mathtt{1}) \neq \mathtt{x}\} & \textit{(assignment)}
\end{array}$$

Note that $(\mathtt{x} = \mathtt{0} \vee \mathtt{x}\&(\mathtt{x}-\mathtt{1}) \neq \mathtt{x}) \equiv$ ***true***. *Repeated application of the rule of composi-*

tion in Figure 2.1 yields

$$\{\mathit{true}\}\ \mathtt{y} := \mathtt{x};\ [\mathtt{y} \neq \mathtt{0}];\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1})\ \{\mathtt{y} \neq \mathtt{x}\}\,, \tag{2.1}$$

completing the proof argument.

Analogously, we can apply forward reasoning. To improve readability, we omit basic logical simplification steps in the logic $\mathcal{L}$ in the following proof.

$$\begin{array}{rl}
\{\mathit{true}\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} = \mathtt{x}\} & \textit{(assignment)} \\
\{\mathit{true}\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} = \mathtt{0} \lor \mathtt{y} = \mathtt{x}\} & \textit{(consequence)} \\
\{\mathtt{y} = \mathtt{0} \lor \mathtt{y} = \mathtt{x}\}\ [\mathtt{y} \neq \mathtt{0}]\ \{\mathtt{y} = \mathtt{x} \land \mathtt{y} \neq \mathtt{0}\} & \textit{(condition)} \\
\{\mathtt{y} = \mathtt{x} \land \mathtt{y} \neq \mathtt{0}\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1})\ \{\exists \mathtt{y}' \,.\, \mathtt{y} = \mathtt{y}'\&(\mathtt{y}'-\mathtt{1}) \land \mathtt{y}' = \mathtt{x} \land \mathtt{y}' \neq \mathtt{0}\} & \textit{(assignment)} \\
\{\mathtt{y} = \mathtt{x} \land \mathtt{y} \neq \mathtt{0}\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1})\ \{\mathtt{y} \neq \mathtt{x}\} & \textit{(consequence)}
\end{array}$$

The intermediate predicates correspond to the strongest postconditions in Table 1.2. The application of the consequence rule weakens the postcondition of `y:=x` *and accounts for the fact that the execution of* $[\mathtt{y} \neq 0]$ *silently terminates if* $\mathtt{y} = 0$ *and never reaches the assertion. Again, repeated use of the composition rule yields the Hoare triple (2.1).*

The consequence rule in Figure 2.1 reflects the additional degree of freedom Hoare logic provides compared to Dijkstra's predicate transformers. Preconditions may be strengthened and postconditions may be weakened. Note that the predicate transformers presented so far are monotonic in their second argument, i.e., it follows from $P \Rightarrow Q$ that $wp(\mathtt{stmt}, P) \Rightarrow wp(\mathtt{stmt}, Q)$, and similarly for *wlp* and *sp*. Reasoning with Hoare logic and the consequence rule in particular gives us no such guarantee. In general,

$$(P \Rightarrow P') \land \{P\}\ \mathtt{stmt}\ \{Q\} \land \{P'\}\ \mathtt{stmt}\ \{Q'\} \quad \Rightarrow \quad (Q \Rightarrow Q')$$

does *not* hold. Note that Hoare triples cannot be classified as monotonic or non-monotonic, since, unlike predicate transformers, they do not represent functions. In Section 2.2 we encounter non-monotonic predicate transformers based on Craig interpolation. The con-

sequences of this property for interpolation-based automated verification algorithms are discussed in Section 2.5.1.

2.2 Craig Interpolation

The predicate transformers presented in the previous section provide merely an upper and a lower bound (in the implication order) for the predicates in Hoare logic proofs. In a consecutive sequence of statements, the pre- and postconditions of each respective Hoare triple are bounded from below by the predicates obtained by a forward analysis based on the strongest postcondition, and bounded from above by the predicates constructed using the weakest precondition during backward analysis. This is illustrated in the following example.

Example 2.2.1. *We continue working in the setting of Example 2.1.1. Table 1.2 lists the weakest preconditions and strongest postconditions for each point in the path*

$$\mathtt{y} := \mathtt{x};\ [\mathtt{y} \neq 0];\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-1);\ \mathtt{assert}(\mathtt{y} \neq \mathtt{x})\,.$$

Figure 2.2 is based on this table and indicates the existence of more than one Hoare logic proof for the safety of the path. At each location in the path, we have a choice of predicates only restricted by the strongest postcondition and weakest precondition of the respective prefix and suffix of the path, respectively. In particular, the proof using the predicate $(\mathtt{y} \leq (\mathtt{x}-1) \wedge \mathtt{x} \neq 0)$ *is a valid Hoare logic proof, as can be shown using the consequence rule:*

$$\begin{array}{rl} \{\mathtt{y}=\mathtt{x} \wedge \mathtt{y} \neq 0\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-1)\ \{\mathtt{y}=\mathtt{x}\&(\mathtt{x}-1) \wedge \mathtt{x} \neq 0\} & \textit{cf. Example 2.1.1} \\ \{\mathtt{y}=\mathtt{x} \wedge \mathtt{y} \neq 0\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-1)\ \{\mathtt{y} \leq (\mathtt{x}-1) \wedge \mathtt{x} \neq 0\} & \textit{(consequence)} \\ \{\mathtt{y}=\mathtt{x} \wedge \mathtt{y} \neq 0\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-1)\ \{\mathtt{y} \neq \mathtt{x}\} & \textit{(consequence)} \end{array}$$

Weakening the postcondition of the first Hoare triple of the proof enables us to show that $\mathtt{y} \leq (\mathtt{x}-1) \wedge \mathtt{x} \neq 0$ *is a valid conclusion (which can be further weakened to* $\mathtt{y} \neq \mathtt{x}$*). The rest of the proof is as in Example 2.1.1.*

Formally, the upper and lower bounds for predicates can be defined in terms of the

$$
\{\textsf{true}\} \quad \texttt{y} := \texttt{x} \quad \begin{matrix} \left\{ \begin{matrix} \texttt{y} = \texttt{0} \lor \\ \texttt{y}\&(\texttt{y}-\texttt{1}) \neq \texttt{x} \end{matrix} \right\} \\ \vdots \\ \{\texttt{x} = \texttt{y}\} \end{matrix} \quad [\texttt{y} \neq \texttt{0}] \quad \begin{matrix} \{\texttt{y}\&(\texttt{y}-\texttt{1}) \neq \texttt{x}\} \\ \vdots \\ \{\texttt{y} = \texttt{x} \land \texttt{y} \neq \texttt{0}\} \end{matrix} \quad \texttt{y} := \texttt{y}\&(\texttt{y}-\texttt{1}) \quad \begin{matrix} \{\texttt{x} \neq \texttt{y}\} \\ \vdots \\ \{\texttt{y} \leq (\texttt{x}-\texttt{1}) \land \texttt{x} \neq \texttt{0}\} \\ \vdots \\ \{\texttt{y} = \texttt{x}\&(\texttt{x}-\texttt{1}) \land \texttt{x} \neq \texttt{0}\} \end{matrix} \quad \texttt{assert}(\texttt{y} \neq \texttt{x})
$$

Figure 2.2: Hoare logic proofs allow a range of predicates

predicate transformers introduced in Definitions 2.1.2 and 2.1.4.

Definition 2.2.1 (Chain Condition). *Let* $\texttt{stmt}_1; \ldots ; \texttt{stmt}_n$ *be a path of length* n *(i.e., a sequence of* n *statements). Furthermore, let* $P_0, \ldots, P_n$ *be a set of* $n+1$ *predicates such that* $\{P_{i-1}\}\ \texttt{stmt}_i\ \{P_i\}$ *is a Hoare triple for all* $i \in \{1..n\}$*. Then, by Definitions 2.1.2 and 2.1.4 it has to hold that*

$$\forall i \in \{1..n\}\,. \qquad P_{i-1} \Rightarrow wlp(\texttt{stmt}_i, P_i) \qquad \wedge \qquad sp(\texttt{stmt}_i, P_{i-1}) \Rightarrow P_i\,.$$

McMillan [McM06] observes that it is possible to obtain predicates satisfying the conditions in Definition 2.2.1 by means of *Craig interpolation*. William Craig's seminal paper [Cra57a] presents the following result:

Theorem 2.2.1 (Craig's Interpolation Theorem). *Let* A *and* B *be two closed formulae in first order predicate logic with equality. Given a valid implication* $A \Rightarrow B$*, there exists an "intermediate" closed formula* I *implied by* A *and implying* B *such that the function and predicate symbols (other than the identity symbol) of* I *occur in* A *as well as in* B*. We call such a formula* I *an interpolant for the pair of formulae* (A, B)*.*

Craig [Cra57a, Cra57b] provides a constructive proof for his theorem. Variants of this theorem have been shown for logics other than first order logic. Furthermore, there are a number of algorithms for computing interpolants for propositional logic [Hua95, Kra97, Pud97, Bus99, DPWK10] and for various combinations of quantifier-free equality logic with uninterpreted functions and linear arithmetic (e.g., [McM05, RSS07, BZM08, FGG$^+$09, GKT09, CGS09]).

Kovács and Voronkov [KV09b] points out that McMillan [McM05] makes slight changes to the notion of an interpolant and introduces inconsistencies with the original terminology (presented in Theorem 2.2.1). For reasons discussed at the end of this section, [McM05] defines an interpolant for an inconsistent pair of formulae (A, B) to be a formula I such that $A \Rightarrow I$ and $B \Rightarrow \neg I$. In the terminology of Theorem 2.2.1, this corresponds to a Craig interpolant for the pair of formulae $(A, \neg B)$.

In this dissertation, we adhere to Craig's original terminology. The interpolation theo-

rem, however, needs to be carefully adapted to our setting. Theorem 2.2.1 does not allow free variables and is therefore not immediately applicable to the quantifier free predicates we use to construct Hoare proofs. Therefore, we treat free program variables in predicates as constants (i.e., uninterpreted nullary functions) in the respective first order language (as suggested in Section 5 of [KV09b]). Moreover, in the context of software verification, it is common to apply a variation of Theorem 2.2.1 which requires only the *uninterpreted* symbols to be shared (cf. [KV09b], Theorem 3). We address this issue in Section 3.2 after specifying the first-order language $\mathcal{L}$ introduced in Section 2.1 in more detail in Section 3.1.

The condition for the predicates P_i $(0 < i < n)$ in Definition 2.2.1 can be formulated alternatively as

$$sp(\mathtt{stmt}_i, P_{i-1}) \Rightarrow P_i \quad \wedge \quad P_i \Rightarrow wlp(\mathtt{stmt}_{i+1}, P_{i+1}) \ . \tag{2.2}$$

Therefore, the interpolant for the pair of formulae ($sp(\mathtt{stmt}_i, P_{i-1})$, $wlp(\mathtt{stmt}_{i+1}, P_{i+1})$) is a valid candidate for P_i. Unfortunately, condition (2.2) defines P_i in terms of P_{i+1} and vice versa and does therefore not enable the actual construction of these interpolants. A slight modification to the condition makes the iterative computation of interpolants possible. Let $\mathtt{stmt}_1; \ldots ; \mathtt{stmt}_n$ be a path of length n. We can derive each P_i $(0 < i < n)$ from its predecessor P_{i-1} by constructing an interpolant for the pair of formulae

$$(\ sp(\mathtt{stmt}_i, P_{i-1}) \ , \ wlp(\mathtt{stmt}_{i+1}; \ldots ; \mathtt{stmt}_n, \ P_n) \ ) \, . \tag{2.3}$$

The resulting sequence of interpolants adheres to the condition in Definition 2.2.1. The condition that $sp(\mathtt{stmt}_i, P_{i-1}) \Rightarrow P_i$ has to hold guarantees that the resulting interpolants can be used to form a sequence of Hoare triples. The condition is introduced in [McM06]. Heizmann et al. [HHP10] calls such a sequence of interpolants *inductive*.

We conclude this section by pointing out that each path $\mathtt{stmt}_1; \ldots ; \mathtt{stmt}_n$ can be transformed into a conjunction of (existentially quantified) predicates representing the respective statements. This is possible because of the correspondence between statements and predicates (noted in Section 2.1 on page 28). The observation is based on the proof of the

following Lemma:

Lemma 2.2.1. *Let* $\pi_1 = \mathtt{stmt}_1; \ldots; \mathtt{stmt}_i$ *and* $\pi_2 = \mathtt{stmt}_{i+1}; \ldots; \mathtt{stmt}_n$ *(with* $0 \leq i \leq n$*, allowing* π_1 *or* π_2 *to be* ε*) be the prefix and the suffix of a path of length* n*, respectively. Then, given a quantifier-free predicate* Q*,* $sp(\pi_1, Q)$ *(and* $wlp(\pi_2, Q)$*, respectively) can be represented as an existentially (universally) quantified predicate in prenex normal form.*

Proof. 1. We prove the first case ($sp(\pi_1, \mathsf{true})$) of Lemma 2.2.1 by induction.

Base case. By definition, $sp(\varepsilon, Q) = Q$. This establishes the base case.

Induction hypothesis. Let π be a path of length $i - 1$. Then, $sp(\pi, Q)$ can be represented as an existentially quantified predicate in prenex normal form.

Induction step. Let $\pi; \mathtt{stmt}$ be a path of length i, and let $P = sp(\pi, Q)$. By the induction hypothesis, P is an existentially quantified predicate in prenex normal form. According to Table 2.1, $sp(\pi; \mathtt{stmt}, Q) = sp(\mathtt{stmt}, P)$. By definition, the predicate R in conditions and assertions is quantifier-free (see Section 2.1). It is therefore straight forward to transform $P \wedge R$ into an equivalent predicate in prenex normal form by means of renaming the quantified variables in P and extending the scope of the quantifier to R. According to Table 2.1, $sp(\mathtt{x} := e, P)$ is $\exists \mathtt{x}' \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{x}']) \wedge P[\mathtt{x}/\mathtt{x}']$, which can be brought into prenex normal form by renaming the quantified variables in P and such that they do not conflict with the free variables in $\exists \mathtt{x}' \,.\, \mathtt{x} = e[\mathtt{x}/\mathtt{x}']$ and by shifting the existential quantifiers of P to the left.

2. To prove the second case ($wlp(\pi_2, Q)$) of Lemma 2.2.1 we use the following theorem:

Theorem 2.2.2 (Predicate Correspondence, [Nel89])**.** *Let* $\mathtt{stmt}$ *be a statement testing or setting no program variables other than* $\mathtt{x}$*. For any predicate* P*, it holds that*

$$wlp(\mathtt{stmt}, P) \equiv \; (\forall \mathtt{x}' \,.\, wlp(\mathtt{stmt}, \mathtt{x} \neq \mathtt{x}') \vee P[\mathtt{x}/\mathtt{x}'])$$

We use Theorem 2.2.2 to show that $wlp(\pi_2, Q)$ can be represented as a universally quantified predicate in prenex normal form.

Base case. By definition, $wlp(\varepsilon, Q) = Q$. This establishes the base case.

Induction hypothesis. Let π be a path of length $i - 1$. Then, $wlp(\pi, Q)$ can be represented as a universally quantified predicate in prenex normal form.

Induction step. Let $\texttt{stmt};\pi$ be a path of length i, and let $P = wlp(\pi, Q)$. By the induction hypothesis, P is a universally quantified predicate in prenex normal form. Then, according to Table 2.1, $wlp(\texttt{stmt};\pi, \mathsf{true}) = wlp(\texttt{stmt}, P)$. By Theorem 2.2.2, $wlp(\texttt{stmt}, P)$ is $\forall \texttt{x}' \,.\, wlp(\texttt{stmt}, \texttt{x} \neq \texttt{x}') \vee P[\texttt{x}/\texttt{x}']$. As in case 1, we make a case distinction for stmt according to Table 2.1.

(a) *Assignment.* By Theorem 2.2.2 and Table 2.1, $wlp(\texttt{x} := e, P)$ is equivalent to $\forall \texttt{x}' \,.\, \neg(\texttt{x}' = e) \vee P[\texttt{x}/\texttt{x}']$. We can rename the quantified variables in P such that they do not conflict with the variables in $\texttt{x}' \neq e$ and move the universal quantifiers of P to the front.

(b) *Condition, Assertion.* Analogously, we obtain $\forall \texttt{x}' \,.\, \neg(R \wedge (\texttt{x} = \texttt{x}')) \vee P[\texttt{x}/\texttt{x}']$ for $wlp([R], P)$ or $wlp(\texttt{assert}(R), P)$, where R is a quantifier-free predicate, which can be brought into prenex normal form as outlined previously.

The argument can be generalised to statements testing or setting more than one program variable by adjusting the predicate $\texttt{x} \neq \texttt{x}'$ in Theorem 2.2.2 accordingly.

□

The reason for presenting this relatively trivial proof in such detail is that it provides valuable insight about the structure of the predicates representing $sp(\pi_1, P)$ and $wlp(\pi_2, Q)$. Let $\pi_1 = \texttt{stmt}_1; \ldots ; \texttt{stmt}_i$ and $\pi_2 = \texttt{stmt}_{i+1}; \ldots ; \texttt{stmt}_n$ be a prefix and a suffix, respectively, of a path of length n. We make the following observations:

1. The existentially quantified variables in $sp(\pi_1, P)$ refer to values of the respective program variables in the past (with respect to the program location following statement

$\texttt{stmt}_i$). Conversely, the universally quantified variables in $wp(\pi_2, Q)$ refer to values of the respective program variables in the future. The free variables in $sp(\pi_1, P)$ and $wp(\pi_2, Q)$ refer to the values in the "current" time-frame. Note that not all free variables in $sp(\pi_1, P)$ must necessarily occur in $wp(\pi_2, Q)$ as well, since π_1 may define variables that are never used in π_2. The "strongest postconditions and weakest preconditions along the path" are therefore *not* necessarily "the strongest and weakest interpolants respectively" as claimed in [McM06].

2. By construction (see case 1 in the proof for Lemma 2.2.1), $sp(\pi_1, Q)$ is an existentially quantified conjunction of predicates, each of which corresponds to one statement in π_1 or to $Q[x/x']$.

3. By construction (see case 2 in the proof for Lemma 2.2.1), $wp(\pi_1, Q)$ is a universally quantified disjunction of predicates, each of which is derived from a statement in π_2 or from Q. Moreover, all predicates derived from program statements occur negated. By negating $wp(\pi_1, Q)$, we can transform the formula into an existentially quantified conjunction of predicates, each of which corresponds to a statement in π_2 or to $\neg Q[x/x']$.

4. Suppose that $sp(\pi_1, P) \Rightarrow wp(\pi_2, Q)$ and let I be an interpolant for the pair of formulae $(\, sp(\pi_1, P), wp(\pi_2, Q)\,)$. Then $sp(\pi_1, P) \Rightarrow I$ and $\neg wp(\pi_2, Q) \Rightarrow \neg I$. Moreover, the formula $sp(\pi_1, P) \wedge \neg wp(\pi_2, Q)$ is unsatisfiable.

The last observation is the motivation for the unconventional definition of Craig interpolants in McMillan [McM05], which is now commonly used in the context of automated verification. Chapter 3 of this dissertation discusses algorithms that are able to derive Craig interpolants from refutation proofs for the conjunction $sp(\pi_1, P) \wedge \neg wp(\pi_2, Q)$. The runtime of these algorithms is polynomial in the size of the refutation proof.

Example 2.2.2. *Let*

$$\pi_1 = \ \mathtt{y} := \mathtt{x};\ \ [\mathtt{y} \neq \mathtt{0}] \quad \textit{and} \quad \pi_2 = \ \mathtt{y} := \mathtt{y}\&(\mathtt{y}-\mathtt{1});\, \mathtt{assert}(\mathtt{x} \neq \mathtt{y})$$

be the prefix and suffix of a path. In order to show that the path is safe, we follow the

instructions outlined in the proof of Lemma 2.2.1 and derive (using Theorem 2.2.2 presented in the proof of Lemma 2.2.1)

$$sp(\mathtt{y} := \mathtt{x};\ \ [\mathtt{y} \neq 0], \textit{true}) \equiv \mathtt{y} = \mathtt{x} \wedge \mathtt{y} \neq 0$$

$$wlp(\mathtt{y} := \mathtt{y}\&(\mathtt{y}-1), \mathtt{x} \neq \mathtt{y}) \equiv \forall \mathtt{y}' \,.\, \mathtt{y}' \neq \mathtt{y}\&(\mathtt{y}-1) \ \vee\ \mathtt{x} \neq \mathtt{y}' \,.$$

We negate $\forall \mathtt{y}' \,.\, \mathtt{y}' \neq \mathtt{y}\&(\mathtt{y}-1) \vee \mathtt{x} \neq \mathtt{y}'$ *and obtain* $\exists \mathtt{y}' \,.\, \mathtt{y}' = \mathtt{y}\&(\mathtt{y}-1) \wedge \mathtt{x} = \mathtt{y}'$*. Observe that the conjunction*

$$\exists \mathtt{y}' \,.\, \mathtt{y} = \mathtt{x} \wedge \mathtt{y} \neq 0 \quad \wedge \quad \mathtt{y}' = \mathtt{y}\&(\mathtt{y}-1) \ \wedge\ \mathtt{x} = \mathtt{y}'$$

is unsatisfiable, as shown by the following refutation proof:

$$\dfrac{\dfrac{\dfrac{\mathtt{y} = \mathtt{x}}{\boxed{\mathtt{y} \leq \mathtt{x}}} \qquad \dfrac{\boxed{\mathtt{y} \neq 0} \qquad \dfrac{\mathtt{y}' = \mathtt{y}\&(\mathtt{y}-1)}{\mathtt{y}' \leq \mathtt{y} - 1}}{\mathtt{y}' < \mathtt{y}}}{\mathtt{y}' < \mathtt{x}} \qquad \mathtt{x} = \mathtt{y}'}{\textit{false}}$$

The predicate $\mathtt{y} \leq \mathtt{x} \wedge \mathtt{y} \neq 0$*, which can be derived from the proof of inconsistency (see Chapter 3), is an interpolant for the pair of formulae*

$$(\, sp(\mathtt{y} := \mathtt{x};\ \ [\mathtt{y} \neq 0], \textit{true}),\ wlp(\mathtt{y} := \mathtt{y}\&(\mathtt{y}-1), \mathtt{x} \neq \mathtt{y})\,) \,.$$

We are now in a position to use an interpolating decision procedure to construct a parametrised predicate transformer. The constraint (2.2) dictates the upper and lower boundaries for the result of a "useful" predicate transformer. We assume the existence of a (partial) function $idp : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ mapping a pair of formulae (A, B) to a respective Craig interpolant (as outlined in Example 2.2.2, details are provided in Chapter 3). Let P be a precondition for a statement `stmt`. Given a weakest acceptable postcondition Q such that $sp(\mathtt{stmt}, P) \Rightarrow Q$ we can define a forward predicate transformer $\overrightarrow{pt}(\mathtt{stmt}, Q, P) \stackrel{\text{def}}{=} idp(sp(\mathtt{stmt}, P), Q)$. Since $sp(\mathtt{stmt}, P)$ establishes a lower bound for the predicate, $\{P\}\ \mathtt{stmt}\ \{\overrightarrow{pt}(\mathtt{stmt}, Q, P)\}$ holds.

However, unlike the strongest postcondition *sp*, predicate transformer $\overrightarrow{pt}$ is not monotonic in its rightmost argument. (Recall the remark about monotonicity and Hoare triples at the end of Section 2.1.) For instance, let P_1 and P_2 be $(\texttt{i} > 1)$ and $(\texttt{i} > 0)$ respectively, and note that $P_1 \Rightarrow P_2$. One can now conceive a decision procedure *idp* such that $\overrightarrow{pt}([\texttt{i} \geq 2], (\texttt{i} > 0), P_1) = (\texttt{i} > 0)$ and $\overrightarrow{pt}([\texttt{i} \geq 2], (\texttt{i} > 0), P_2) = (\texttt{i} \geq 2)$. Since $(\texttt{i} \geq 2) \Rightarrow (\texttt{i} > 0)$, $\overrightarrow{pt}$ is not monotonic. We discuss the drawbacks of non-monotonic predicate transformers in Section 2.5.1.

2.3 Programs, Reachability Trees, and Counterexamples

The predicate transformers defined in Section 2.1 define the semantics of statements and sequences of statements (paths, respectively). These elements provide the basis for defining the semantics of programs. In this section, we define the notion of programs in terms of control-flow automata [HJM+02] (which are essentially equivalent to the program graphs of Cousot [CC79]). We introduce the concept of reachability trees [HK76] and formalise the notion of a counterexample.

We use control-flow automata (CFA) to represent programs. A CFA comprises a finite number of nodes $N \subseteq \mathbb{N}$ and a set of directed edges $E \subseteq (N \times N)$. Each CFA contains one designated entry node ➝○, which has no predecessors, and one designated exit node ◎, which has no successors. Each edge $\langle i, j \rangle \in E$ connects two nodes and is annotated with a statement $\texttt{stmt}_{\langle i,j \rangle}$. Overloading our notation, we use the term “path” to refer to a path in the graph of the CFA as well as to the corresponding sequence of statements. Figure 1.1 in Section 1.3.1 shows a specimen of a CFA representing a program.

The semantics of statements is defined in Section 2.1. We extend this definition to programs using the deductive semantics of programs presented in [CC79]. Given a choice of more than one successor node, we assume that the successor is selected non-deterministically: The implementation “clairevoyantly” [Nel89] chooses a condition which succeeds.

The strongest postcondition *sp* (see Table 2.1) determines the forward semantics (Section 3.1 of [CC79]) of a program. The states reachable at each node of a program are determined by a *merge over all paths analysis*.

Definition 2.3.1 (Merge Over All Paths Analysis [CC79])**.** *Let the concrete domain of program states $\mathcal{C}$ be a complete lattice represented by the lattice of first-order logic predicates $\mathcal{P}(\Rightarrow, \textsf{false}, \textsf{true}, \vee, \wedge)$ of the language $\mathcal{L}$. The merge over all paths $MOP(sp, \textsf{true})$ for a node i in a CFA (N, E) is the predicate P_i $(i \in N)$ defined by*

$$P_i = \bigvee_{\pi_i \in \Pi(\rightarrow\!\bigcirc, i)} sp(\pi_i, \textsf{true}) \,,$$

where $\Pi(\rightarrow\!\bigcirc, i)$ denotes the (not necessarily finite) set of paths from $\rightarrow\!\bigcirc$ to the node i. The predicate P_i characterises the set of states reachable at node i.

The predicate transformers for the statements of the program correspond to a system of equations. For each node $i \neq \rightarrow\!\bigcirc$, P_i is determined by the join over the strongest postconditions of the statements labelling the respective incoming edges. Formally,

$$\begin{cases} P_{\rightarrow\bigcirc} = \textsf{true} \\ P_i \ = \bigvee_{j \in \{n \mid \langle n,m\rangle \in E \,\wedge\, m=i\}} sp(\texttt{stmt}_{\langle j,i\rangle}, P_j) \quad \text{if } i \in N \setminus \{\rightarrow\!\bigcirc\} \end{cases} . \tag{2.4}$$

The merge over all paths analysis (Definition 2.3.1) corresponds to the least fixed point of this system of equations [CC79]. Tarski's theorem [Tar55] in combination with the monotonicity of the predicate transformer *sp* warrants the existence of this fixed point. If we evaluate the equations (2.4) in the execution order imposed by the directed edges of the CFA, we obtain a reachability tree:

Definition 2.3.2 (Reachability Tree)**.** *A reachability tree T of a CFA (N, E) for the predicate transformer sp is a directed tree-shaped graph formed by a collection of annotated nodes N_T and a set of edges E_T. Each node in N_T is represented by a tuple comprising a unique time-stamp $t \in \mathbb{N}$ (numbering the nodes in order of creation) and a node $i \in N$. Furthermore, each node (t, i) is annotated with a predicate $P_{(t,i)}$. The following conditions hold:*

- *The set N_T contains exactly one designated root node $\langle 0, \rightarrow\!\bigcirc\rangle$ with in-degree zero, which is annotated with the predicate* **true***.*
- *For every edge $\langle (t_1, i), (t_2, j)\rangle \in E_T$ it holds that*

– $t_1 < t_2$,

– $\langle i, j \rangle \in E$, *and*

– $sp(\texttt{stmt}_{\langle i,j \rangle}, P_{(t_1,i)}) = P_{(t_2,j)}$.

A node (t_1, i) *in* N_T *is fully expanded if for every edge* $\langle i, j \rangle \in E$ *there exists a corresponding edge* $\langle (t_1, i), (t_2, j) \rangle \in E_T$*. We say that a node in* (t, i) *is unreachable if* $P_{(t,i)} \Rightarrow \mathsf{false}$ *and reachable otherwise.*

A reachability tree represents a symbolic simulation of finite-length paths of a program. This approach is sufficient to falsify programs, since counterexamples to reachability properties (such as assertions) are paths of finite length.

Definition 2.3.3 (Counterexample)**.** *Let* T *be a reachability tree of a CFA (for the predicate transformer sp) containing a node* i *with an outgoing edge labelled* $\texttt{assert}(R)$*. A counterexample to this assertion is a path* π *of* T *such that*

- π *starts at* $(0, \rightarrow\!\bigcirc)$,
- π *is incident upon a node* (t, i) *of* T*, and*
- $\neg(sp(\pi, \mathsf{true}) \Rightarrow R)$ *is satisfiable.*

It follows immediately from Lemma 2.2.1 that $\neg(sp(\pi, \mathsf{true}) \Rightarrow R)$ can be represented as an existentially quantified formula in prenex normal form. Therefore, the third condition in Definition 2.3.3 can be checked by means of a satisfiability checker for $\mathcal{L}$. A breadth-first search algorithm is able to provide paths satisfying condition one and two (assuming such paths exist). Automated verification techniques exploring all paths up to a predetermined length fall into the category of bounded model checking (BMC) [BCCZ99]. We refer the reader to our survey paper [DKW08] for a detailed discussion of BMC-based software verification.

BMC is unable to provide safety guarantees beyond the fraction of the state space reachable within a limited number of execution steps. A verification algorithm computing the least fixed point for the equation system (2.4) (or a sound approximation thereof) lifts this restriction. Let P_i denote the predicate corresponding to the merge over all paths for

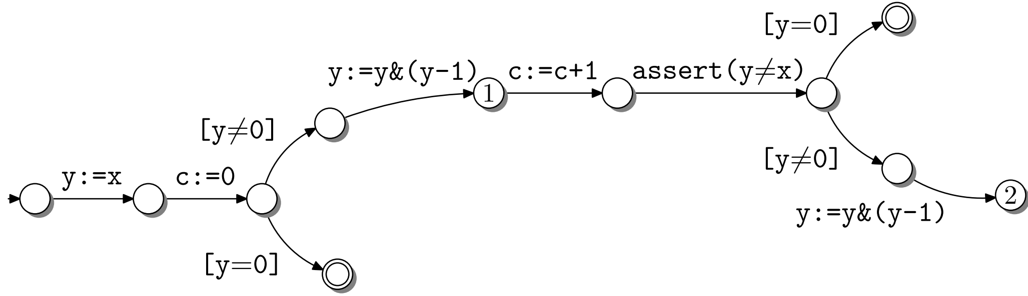


Figure 2.3: A reachability tree for the program in Figure 1.1. The annotations are omitted.

the node i of a CFA (N, E). A program is safe if for each node $i \in N$ and all its outgoing edges labelled $\texttt{assert}(R)$, it holds that $P_i \Rightarrow R$. The safety of a program can therefore be verified by computing the least fixed point for the equation system (2.4).

In general, however, no sound and complete algorithm can achieve this goal (due to Turing's undecidability result in discussed in the introduction [Tur36]). Consider an algorithm that expands the leaf nodes of the reachability tree until a fixed point for

$$P_{\to\bigcirc} = \textsf{true} \qquad \text{and} \qquad P_i = \bigvee\{P_{(t,j)} \mid (t,j) \in E_T \wedge i = j\} \tag{2.5}$$

is reached for each node i in the CFA. Depending on the semantics of the programming language (or more specifically, the underlying first-order language $\mathcal{L}$), this process may not terminate.

Example 2.3.1. *Figure 2.3 shows a reachability tree (omitting the annotations of nodes) for the CFA in Figure 1.1. All nodes in this tree except the leaf-node ② are fully expanded. The nodes labelled ① and ② in Figure 2.3 represent two nodes (t_1, i) and (t_2, i) (where $t_1 < t_2$), respectively, both deriving from a single node i of the CFA in Figure 1.1. From Section 1.3.1, we recall that*

$$P_{(t_1,i)} = (\texttt{y} = (\texttt{x}\&(\texttt{x}-\texttt{1})) \wedge (\texttt{x} \neq 0)) \qquad \textit{and}$$
$$P_{(t_2,i)} = (\texttt{x}\&(\texttt{x}-1) \neq 0) \wedge (\texttt{x} \neq 0) \wedge \texttt{y} = (\texttt{x}\&(\texttt{x}-1))\&((\texttt{x}\&(\texttt{x}-1))-1)\,.$$

Since $P_{(t_2,i)} \Rightarrow \neg P_{(t_1,i)}$, node (t_2, i) has to be expanded further to reach a fixed point for P_i. For variables taking values in a bounded domain, the equations (2.5) converge after a

finite number of expansion steps. If, however, we model program variables using unbounded integers, the sequence of predicates we obtain diverges.

The following section discusses a technique that enables us to avoid the non-termination observed in Example 2.3.1 at the cost of accuracy.

2.4 Predicate Abstraction

This section discusses abstraction as a means to construct approximations of reachability trees. In particular, we focus on predicate abstraction [GS97], which is an instance of Cousot's abstract interpretation [CC77] framework. Predicate abstraction requires the pre- and postconditions tracked at each node of a reachability tree to be Boolean combinations of a finite set of predetermined predicates, resulting in a finitary abstraction of the concrete domain $\mathcal{C}$ of a program. Accordingly, the predicate transformers of Section 2.1 are replaced by approximate counterparts. The finitary nature of the abstract domain prevents the divergence observed in Example 2.3.1 and enables the effective computation of invariants.

The idea to analyse abstractions of programs is motivated by the observation that, in order to prove the safety of a program, it is not always necessary to compute the exact set of reachable states of a program (or the merge over all paths $MOP(sp, \textsf{true})$, respectively).

Example 2.4.1. *The inductive invariant* $\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)$ *is an over-approximation of the set of states reachable at the node right before the assertion in Figure 1.1. To see this, consider the assignment* $\{\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 2\}$*, which satisfies the invariant but represents a state unreachable at the respective program location.*

2.4.1 Approximating Predicate Transformers

In the terminology of Cousot's abstract interpretation framework, a predicate Q "approximates" a predicate P if $P \Rightarrow Q$ (cf. [CC79], Definition 4.0.1). For instance, the inductive invariant $\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)$ in Example 2.4.1 approximates $\mathtt{y} = (\mathtt{x} \& (\mathtt{x} - \mathtt{1})) \wedge (\mathtt{x} \neq 0)$. The predicate $\mathtt{y} = (\mathtt{x} \& (\mathtt{x} - \mathtt{1})) \wedge (\mathtt{x} \neq 0)$ does not occur in the safety argument for the example in Section 1.3.1 and is therefore not required to prove the program safe. By definition, the number of predicates representing the merge over all paths is finite. Therefore, if a

program is safe, then there must be a finite set of predicates which is sufficient to construct a respective Hoare proof. Let $\mathcal{P} \subseteq \mathcal{L}$ be a finite set of quantifier-free predicates which track selected facts about the program and let $\widehat{\mathcal{L}}_{\mathcal{P}}$ denote the Boolean closure of $\mathcal{P}$ (i.e., the language of all well-formed predicates formed using $\mathcal{P}$ and the operators $\wedge$,$\vee$, and $\neg$). Note that there are only finitely many semantically distinct elements in $\widehat{\mathcal{L}}_{\mathcal{P}}$.

Predicate abstraction amounts to approximating the pre- and postconditions of Hoare triples using predicates of the language $\widehat{\mathcal{L}}_{\mathcal{P}}$. Consequently, the choice of $\mathcal{P}$ determines the accuracy of the analysis.

Given a predicate $P \in \widehat{\mathcal{L}}_{\mathcal{P}}$, it is not necessarily the case that $sp(\texttt{stmt}, P) \in \widehat{\mathcal{L}}_{\mathcal{P}}$, i.e., the language $\widehat{\mathcal{L}}_{\mathcal{P}}$ is not necessarily closed under the application of predicate transformers. In order to be able to perform a merge over all paths analysis in the domain defined by $\widehat{\mathcal{L}}_{\mathcal{P}}$, we need to define an approximate counterpart to the strongest postcondition.

Definition 2.4.1 (Correct Upper Approximation ([CC79], Theorem 7.1.0.2))**.** *Let $\widehat{pt}$ be a (not necessarily monotonic) predicate transformer for a language $\widehat{\mathcal{L}} \subseteq \mathcal{L}$. The mapping $\widehat{pt}$ is a correct upper approximation of the strongest postcondition if and only if*

$$\forall P \in \widehat{\mathcal{L}} \,.\, sp(\texttt{stmt}, P) \Rightarrow \widehat{pt}\,(\texttt{stmt},\ P)\,. \tag{2.6}$$

Equivalently, $\{P\}$ $\texttt{stmt}$ $\{\,\widehat{pt}\,(\texttt{stmt},\ P)\,\}$ must hold (according to Definition 2.1.4).

(Note that the parametrised predicate transformer $\overrightarrow{pt}$ defined at the end of Section 2.2 is such a correct upper approximation.) This condition guarantees that the merge over all paths analysis $MOP(\widehat{pt}, \mathsf{true})$ yields an over-approximation of the set of reachable states determined by $MOP(sp, \mathsf{true})$, i.e., it holds that $MOP(sp, \mathsf{true}) \Rightarrow MOP(\widehat{pt}, \mathsf{true})$. While this is sufficient to guarantee the soundness of a safety analysis, Definition 2.4.1 does not guarantee that the approximate predicate transformer $\widehat{pt}$ yields the "best" possible result.

Definition 2.4.2 (Best Correct Upper Approximation)**.** *Given a finite set of predicates $\mathcal{P}$, let $\widehat{sp}(\texttt{stmt}, P)$ be the strongest predicate $Q \in \widehat{\mathcal{L}}_{\mathcal{P}}$ such that $\{P\}$ $\texttt{stmt}$ $\{Q\}$ holds for all predicates $P \in \widehat{\mathcal{L}}_{\mathcal{P}}$. Then (according to Theorem 7.2.0.2 in [CC79])*

1. *$\widehat{sp}$ is monotonic in its second argument and*

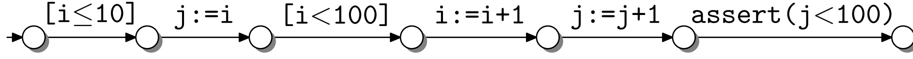


Figure 2.4: A path reaching the assertion of the CFA in Figure 1.7

2. $\widehat{sp}$ is the best correct upper approximation of sp in $\widehat{\mathcal{L}}_{\mathcal{P}}$.

Observation 1 in Definition 2.4.2 warrants the existence of the least fixed point for the system of equations for $\widehat{sp}$ defined analogously to the equations (2.4) in Section 2.3. Moreover, this fixed point is effectively computable due to the fact that the number of semantically distinct predicates in $\widehat{\mathcal{L}}_{\mathcal{P}}$ is finite.[4] Observation 2 in Definition 2.4.2 guarantees that $MOP(\widehat{sp}, \mathsf{true})$ is the best approximation of $MOP(sp, \mathsf{true})$ in $\widehat{\mathcal{L}}_{\mathcal{P}}$.

While the definition of best correct upper approximation resembles the definition of the strongest postcondition (cf. Definition 2.1.4), the predicate transformer $\widehat{sp}$ lacks the simple syntactic construction rules (see Table 2.1 in Section 2.1) of its concrete counterpart. In order to compute $\widehat{sp}(\mathtt{stmt}, P)$ for a predicate $P \in \widehat{\mathcal{L}}_{\mathcal{P}}$, we need to identify the best approximation of $sp(\mathtt{stmt}, P)$ in $\widehat{\mathcal{L}}_{\mathcal{P}}$. For this purpose, we fix a canonical representation for the predicates in $\widehat{\mathcal{L}}_{\mathcal{P}}$. Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be the set of predicates for $\widehat{\mathcal{L}}_{\mathcal{P}}$. A *minterm* over $\mathcal{P}$ is a conjunction $\bigwedge_{i=1}^{n} L_i$ of literals L_i (where $L_i \in \{P_i, \neg P_i\}$). Each minterm represents one of 2^n equivalence classes, which form a partitioning of the concrete state space $\mathcal{C}$. The best approximation of a predicate Q corresponds to the minimal set of partitions covering the states Q represents. Formally, for a fixed set of predicates $\mathcal{P} = \{P_1, \ldots, P_n\}$, the best approximation of a predicate $Q \in \mathcal{L}$ in $\widehat{\mathcal{L}}_{\mathcal{P}}$ is

$$\bigvee \left\{ M \mid M = \bigwedge_{i=1}^{n} L_i \ \text{ (where } L_i \in \{P_i, \neg P_i\}\text{) and } Q \wedge M \not\Rightarrow \mathsf{false} \right\} . \tag{2.7}$$

The following example illustrates reasoning with the abstract predicate transformer $\widehat{sp}$.

Example 2.4.2. *Consider the program in Figure 1.7. Figure 2.4 shows a path reaching the assertion of this program. Note that that the path does not violate the assertion (cf. Section 1.3.2). Now, let $\mathcal{P} = \{(\mathtt{i} = \mathtt{j}), (\mathtt{i} < 100)\}$. We aim to establish the safety of the path by showing that $(\mathtt{i} = \mathtt{j}) \wedge (\mathtt{i} < 100)$ holds right before the assertion by symbolically*

[4] We implicitly assume that semantically equivalent predicates can be identified, i.e., that $\widehat{\mathcal{L}}_{\mathcal{P}}$ is decidable.

simulating the path in $\widehat{\mathcal{L}}_{\mathcal{P}}$*. We observe that*

$$sp([\mathtt{i} \leq 10], \textit{\textsf{true}}) = \mathtt{i} \leq 10\,.$$

The best approximation (2.7) of $[\mathtt{i} \leq 10]$ *in* $\widehat{\mathcal{L}}_{\mathcal{P}}$ *is*

$$(\mathtt{i} = \mathtt{j} \wedge \mathtt{i} < 100) \vee\ (\neg(\mathtt{i} = \mathtt{j}) \wedge \mathtt{i} < 100)\,.$$

Accordingly, we obtain $\widehat{sp}([\mathtt{i} \leq 10], \textit{\textsf{true}}) = (\mathtt{i} < 100)$*. Analogously, we derive*

$$\begin{array}{lcll} \widehat{sp}(\mathtt{j} := \mathtt{i}, \mathtt{i} < 100) & = & (\mathtt{i} < 100) \wedge (\mathtt{i} = \mathtt{j}) & \textit{and} \\ \widehat{sp}([\mathtt{i} < 100], (\mathtt{i} < 100) \wedge (\mathtt{i} = \mathtt{j})) & = & (\mathtt{i} < 100) \wedge (\mathtt{i} = \mathtt{j}) & \end{array}$$

from the strongest postcondition sp of the respective predicates. Note that we do not loose any more precision in this simulation step. In the next step, however, we obtain

$$sp(\mathtt{i} := \mathtt{i} + \mathtt{1}, (\mathtt{i} < 100) \wedge (\mathtt{i} = \mathtt{j})) = (\mathtt{i} \leq 100) \wedge (\mathtt{i} = \mathtt{j} + 1)\,.$$

The best approximation of $(\mathtt{i} \leq 100) \wedge (\mathtt{i} = \mathtt{j} + 1)$ *in* $\widehat{\mathcal{L}}_{\mathcal{P}}$ *is*

$$(\mathtt{i} < 100) \wedge \neg(\mathtt{i} = \mathtt{j}) \vee \neg(\mathtt{i} < 100) \wedge \neg(\mathtt{i} = \mathtt{j})\,,\ \textit{or equivalently}\ \neg(\mathtt{i} = \mathtt{j})\,.$$

Finally, the strongest postcondition $sp(\mathtt{j} := \mathtt{j} + \mathtt{1}, (\mathtt{i} \neq \mathtt{j})) = (\mathtt{i} + 1 \neq \mathtt{j})$ *and its approximation* ***true*** *is insufficient to establish the required assertion* $(\mathtt{i} = \mathtt{j}) \wedge (\mathtt{i} < 100)$.

Note that the simulation of the path in Figure 2.4 in $\widehat{\mathcal{L}}_{\mathcal{P}}$ corresponds to a path in the reachability tree in which the approximate predicate transformer $\widehat{sp}$ has been substituted for its concrete counterpart *sp* (see Definition 2.3.2). The prefix

$$\pi = [\mathtt{i} \leq 10];\ \mathtt{j} := \mathtt{i};\ [\mathtt{i} < 100];\ \mathtt{i} := \mathtt{i} + \mathtt{1};\ \mathtt{j} := \mathtt{j} + \mathtt{1}$$

of the path in Figure 2.4 satisfies the first two conditions of the Definition 2.3.3 of a counterexample. Furthermore, the formula $\neg(\widehat{sp}(\pi, \textsf{true}) \Rightarrow (\mathtt{j} < 100))$ is satisfiable (since

$\widehat{sp}(\pi, \mathsf{true}) = \mathsf{true}$). We say that the path constitutes an abstract counterexample.

Definition 2.4.3 (Abstract Reachability Tree, Abstract Counterexample)**.** *Abstract reachability trees (ART) and an abstract counterexamples (for a correct upper approximation $\widehat{pt}$ of the strongest postcondition) are defined as in Definition 2.3.2 and Definition 2.3.3, respectively, with the only difference that the predicate transformer sp is replaced with its abstract counterpart $\widehat{pt}$ wherever it occurs in the respective definition.*

Whenever we wish to distinguish more clearly between reachability trees and counterexamples and their abstract counterparts, we refer to the artefacts originally defined in Definition 2.3.2 and Definition 2.3.3 as *concrete* reachability trees and *concrete* counterexamples, respectively.

Definition 2.4.4 (Spurious Counterexample)**.** *An abstract counterexample is spurious if the corresponding path in the respective concrete reachability tree does not constitute a concrete counterexample.*

For instance, the path in Example 2.4.2 is a spurious counterexample, since $sp(\pi, \mathsf{true}) = (\mathtt{j} \leq 11) \wedge (\mathtt{i} \leq 11) \wedge (\mathtt{i} < 101)$ and therefore the assertion $\mathtt{j} < 100$ holds. The reason is that the predicates obtained by means of the approximate predicate transformer are not strong enough (with respect to the chain condition in Definition 2.2.1) to prove $\{\mathsf{true}\}\ \pi\ \{\mathtt{j} < 100\}$. Intuitively, adding any sequence of predicates that satisfies the chain condition in Definition 2.2.1 to $\mathcal{P}$ suffices to eliminate the spurious counterexamples. In Section 2.5 we discuss techniques to refine approximations in more detail.

2.4.2 Abstract Domains and Transition Relations

The minterms in Formula (2.7) partition the state space into a finite number of equivalence classes. Given a set of predicates $\mathcal{P} = \{P_1, \ldots P_n\}$, each minterm $\bigwedge_{i=1}^{n} L_i$ (with $L_i \in \{P_i, \neg P_i\}$) can be alternatively represented by means of $\bigwedge_{i=1}^{n} (\mathtt{x}_i \Leftrightarrow P_i)$ and a fixed valuation of the Boolean variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$. Accordingly, each equivalence class (or minterm over $\mathcal{P}$, respectively) corresponds to a valuation of the variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ (or the corresponding minterm $\bigwedge_{i=1}^{n} L_i$, $L_i \in \{\mathtt{x}_i, \neg\mathtt{x}_i\}$, if we maintain a symbolic representation). We refer to such a valuation as an *abstract state* $s \in \mathbb{B}^n$ and use the the language

of propositional logic $\mathcal{L}_{\mathbb{B}}$ as a symbolic representation for sets of abstract states. The state space induced by the Boolean variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ represents a finite *abstract domain* [GS97, CC77]. Intuitively, the best approximation (2.7) associates a set of abstract states to each predicate $Q \in \mathcal{L}$. This intuition is captured formally by the following definition:

Definition 2.4.5 (Boolean Approximation [BPR03]). *Let $\mathcal{P} = \{P_1, \ldots P_n\}$ be a fixed set of predicates. The abstraction function $\alpha_{bool} : \mathcal{L} \to \mathcal{L}_{\mathbb{B}}$ mapping a predicate $Q \in \mathcal{L}$ to a set of abstract states is defined as*

$$\alpha_{bool}(Q) \stackrel{\text{def}}{=} \bigvee \left\{ \bigwedge_{i=1}^{n} (\mathtt{x}_i \Leftrightarrow b_i) \,|\, \{b_1, \ldots, b_n\} \in \mathbb{B}^n \text{ such that } Q \wedge \left(\bigwedge_{i=1}^{n} b_i \Leftrightarrow P_i \right) \not\Rightarrow \textit{false} \right\},$$

i.e., $\alpha_{bool}(Q)$ is a propositional predicate over the Boolean variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$.

Conversely, the concretisation function $\gamma_{bool} : \mathcal{L}_{\mathbb{B}} \to \mathcal{L}$ for a propositional predicate over the variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ is defined as

$$\gamma_{bool}(Q) \stackrel{\text{def}}{=} \left(\exists \mathtt{x}_1, \ldots, \mathtt{x}_n \,.\, Q \wedge \bigwedge_{i=1}^{n} \mathtt{x_i} \Leftrightarrow P_i \right).$$

Note that $\gamma_{\text{bool}}(Q)$ is a predicate in the language $\widehat{\mathcal{L}_{\mathcal{P}}}$ if we eliminate the existential quantification by enumerating the satisfying assignments of $(\exists \mathtt{x}_1, \ldots, \mathtt{x}_n \,.\, Q)$. We immediately obtain an alternative definition of the best upper approximation $\widehat{sp}$ of the predicate transformer sp:

$$\forall P \in \widehat{\mathcal{L}_{\mathcal{P}}} \,.\, \widehat{sp}(\mathtt{stmt}, P) = \gamma_{\text{bool}}(\alpha_{\text{bool}}(sp(\mathtt{stmt}, P))) \tag{2.8}$$

The best approximate predicate transformer $\widehat{sp}$ uniquely determines an abstract counterpart $\widehat{\mathtt{stmt}}$ for each program statement $\mathtt{stmt}$. Analogously to concrete statements, abstract statements can be represented as predicates over initial and final values, i.e., as relations over two sets of variables $\{\grave{\mathtt{x}}_1, \ldots, \grave{\mathtt{x}}_n\}$ and $\{\acute{\mathtt{x}}_1, \ldots, \acute{\mathtt{x}}_n\}$ (cf. page 28 in Section 2.1). We are now in a position to define this *abstract transition relation* in terms of the pair of functions $\alpha_{\text{bool}}(Q)$ and $\gamma_{\text{bool}}(Q)$.

Definition 2.4.6 (Abstract Transition Relation)**.** *Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a fixed set of predicates and let α_{bool} and γ_{bool} be the corresponding abstraction and concretisation functions. Let $sp_{\mathsf{bool}}(\mathtt{stmt}, P) = \alpha_{\mathsf{bool}}(sp(\mathtt{stmt}, \gamma_{\mathsf{bool}}(P))) : \mathcal{L}_{\mathbb{B}} \to \mathcal{L}_{\mathbb{B}}$ be a predicate transformer for the abstract domain of propositional predicates over $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$. The abstract transition relation $\widehat{\mathtt{stmt}} \subseteq (\mathbb{B}^n \times \mathbb{B}^n)$ for a program statement $\mathtt{stmt}$ is defined as*

$$\widehat{\mathtt{stmt}}(\grave{\mathtt{x}}_1, \ldots, \grave{\mathtt{x}}_n,\, \acute{\mathtt{x}}_1, \ldots, \acute{\mathtt{x}}_n) \stackrel{\text{def}}{=}$$
$$\bigvee \left\{ \exists \mathtt{x}_1, \ldots, \mathtt{x}_n \,.\, \bigwedge_{i=1}^{n} \mathtt{x}_i \Leftrightarrow \grave{\mathtt{x}}_i \,\wedge \right.$$
$$\left. sp_{\mathsf{bool}}(\mathtt{stmt}, \bigwedge_{i=1}^{n} \mathtt{x}_i \Leftrightarrow b_i)[\mathtt{x_1}/\acute{\mathtt{x}}_\mathtt{1}][\ldots][\mathtt{x_n}/\acute{\mathtt{x}}_\mathtt{n}] \mid \{b_1, \ldots, b_n\} \in \mathbb{B}^n \right\} \tag{2.9}$$

The core of the Relation (2.9) in Definition 2.4.6 is formed by the abstract predicate transformer $sp_{\mathsf{bool}}(\mathtt{stmt}, P) = \alpha_{\mathsf{bool}}(sp(\mathtt{stmt}, \gamma_{\mathsf{bool}}(P)))$ [BPR03]. It maps each set of abstract states characterised by the predicate $P \in \mathcal{L}_{\mathbb{B}}$ to a set of abstract successor states. The predicate transformer sp_{bool} enables a transition from an abstract state represented by a minterm $M_i \in \mathcal{L}_{\mathbb{B}}$ into an abstract state characterised by a minterm $M_j \in \mathcal{L}_{\mathbb{B}}$ if and only if the formula $sp(\mathtt{stmt}, \gamma_{\mathsf{bool}}(M_i)) \wedge \gamma_{\mathsf{bool}}(M_j)$ is satisfiable. Intuitively, this means that there exists a pair of concrete states s_i and s_j in $\gamma_{\mathsf{bool}}(M_i)$ and $\gamma_{\mathsf{bool}}(M_j)$, respectively, such that s_j is reachable from s_i by executing $\mathtt{stmt}$. Formally, for $P = \gamma_{\mathsf{bool}}(M_i)$ and $Q = \gamma_{\mathsf{bool}}(M_j)$, $M_j \Rightarrow sp_{\mathsf{bool}}(\mathtt{stmt}, M_i)$ if and only if

$$\exists s, s' \,.\, sp(\mathtt{stmt}, P(s)) \wedge Q(s')\,. \tag{2.10}$$

This follows immediately from the definition of α_{bool} (Definition 2.4.5). We can rewrite Relation (2.10) as

$$\neg\forall s, s' \,.\, \neg \left(sp(\mathtt{stmt}, P(s)) \wedge Q(s')\right) \quad \text{which in turn is equivalent to}$$
$$\neg\forall s, s' \,.\, sp(\mathtt{stmt}, P(s)) \Rightarrow \neg Q(s')\,.$$

Stated as a Hoare triple, this yields

$$
\begin{aligned}
&\neg(\ \{P\}\ \texttt{stmt}\ \{\neg Q\}\ ) \qquad \text{or} \\
&\neg(\ \{\gamma_{\mathsf{bool}}(M_i)\}\ \texttt{stmt}\ \{\neg\gamma_{\mathsf{bool}}(M_j)\}\ )\ ,
\end{aligned}
\tag{2.11}
$$

respectively. Intuitively, a transition between the abstract states M_i and M_j is enabled unless it can be explicitly ruled out by means of a Hoare logic proof. Therefore, if a state is reachable in the concrete program, then the corresponding abstract state is also reachable in the abstraction, meaning that reachability is preserved. This abstraction technique is known as existential abstraction [CGL94].

Example 2.4.3. *We continue working in the setting of Example 2.4.2. Let* $\texttt{x}_1$ *and* $\texttt{x}_2$ *be the Boolean variables corresponding to* $(\texttt{i} = \texttt{j})$ *and* $(\texttt{i} < 100)$*, respectively. Consider the conditional statement* $[\texttt{i} < 10]$*. Definition 2.4.6 requires us to compute the set of abstract successor states for all minterms over* $\texttt{x}_1$ *and* $\texttt{x}_2$*. We derive*

$$
\begin{aligned}
&sp([\texttt{i} \leq 10], \texttt{i} \neq \texttt{j} \wedge \texttt{i} \geq 100) = \textsf{\textit{false}} && \gamma_{\textsf{\textit{bool}}}(\textsf{\textit{false}}) = \textsf{\textit{false}} \\
&sp([\texttt{i} \leq 10], \texttt{i} \neq \texttt{j} \wedge \texttt{i} < 100) = \texttt{i} \neq \texttt{j} \wedge \texttt{i} \leq 10 && \gamma_{\textsf{\textit{bool}}}(\texttt{i} \neq \texttt{j} \wedge \texttt{i} \leq 10) = \neg\texttt{x}_1 \wedge \texttt{x}_2 \\
&sp([\texttt{i} \leq 10], \texttt{i} = \texttt{j} \wedge \texttt{i} \geq 100) = \textsf{\textit{false}} && \gamma_{\textsf{\textit{bool}}}(\textsf{\textit{false}}) = \textsf{\textit{false}} \\
&sp([\texttt{i} \leq 10], \texttt{i} = \texttt{j} \wedge \texttt{i} < 100) = \texttt{i} \neq \texttt{j} \wedge \texttt{i} \leq 10 && \gamma_{\textsf{\textit{bool}}}(\texttt{i} = \texttt{j} \wedge \texttt{i} \leq 10) = \texttt{x}_1 \wedge \texttt{x}_2
\end{aligned}
$$

This yields the relation

$$
(\neg\grave{\texttt{x}}_1 \wedge \grave{\texttt{x}}_2 \wedge \neg\acute{\texttt{x}}_1 \wedge \acute{\texttt{x}}_2)\ \vee\ (\grave{\texttt{x}}_1 \wedge \grave{\texttt{x}}_2 \wedge \acute{\texttt{x}}_1 \wedge \acute{\texttt{x}}_2)\ ,
$$

which can be written as $(\grave{\texttt{x}}_1 \Leftrightarrow \acute{\texttt{x}}_1) \wedge \grave{\texttt{x}}_2 \wedge \acute{\texttt{x}}_2$.

The enumeration of all 2^n minterms over the predicates $\{P_1, \ldots, P_n\}$ and the respective calls to the decision procedure for $\mathcal{L}$ required for the computation of α_{bool} are extremely computationally expensive. In practice, it is therefore common to compute an upper approximation of the most accurate transition relation. Cartesian abstraction [GS97, BPR03] is one such approximation.

Definition 2.4.7 (Cartesian Approximation [GS97])**.** *Let $\mathcal{P} = \{P_1, \ldots P_n\}$ be a fixed set of predicates and $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ the corresponding set of Boolean variables. The Cartesian abstraction function $\alpha_{\mathsf{cart}} : \mathcal{L} \to \mathcal{L}_{\mathbb{B}}$ mapping a predicate $Q \in \mathcal{L}$ to a set of abstract states is defined as*

$$\alpha_{\mathsf{cart}}(Q) \stackrel{\text{def}}{=} \bigwedge (\{\mathtt{x}_i \mid Q \Rightarrow P_i, 1 \le i \le n\} \cup \{\neg\mathtt{x}_i \mid Q \Rightarrow \neg P_i, 1 \le i \le n\}) \ .$$

Cartesian abstraction considers each predicate P_i independently and may therefore eliminate correlations between predicates which would have been preserved by α_{bool}. For instance, let $\mathtt{x}_1$ and $\mathtt{x}_2$ correspond to $(\mathtt{i} = 0)$ and $(\mathtt{i} > 10)$, respectively. Then $\alpha_{\mathsf{cart}}(\mathtt{i} < 100)$ is true, allowing the valuation $\{\mathtt{x}_1 \mapsto \mathsf{true}, \mathtt{x}_2 \mapsto \mathsf{true}\}$ despite the fact that $\gamma_{\mathsf{bool}}(\mathtt{x}_1 \wedge \mathtt{x}_2)$ yields the unsatisfiable formula $(\mathtt{i} = 0) \wedge (\mathtt{i} > 10)$.

The definition of the abstract transition function for Cartesian abstraction resembles Definition 2.4.6 except that α_{cart} replaces α_{bool}. Accordingly, the predicate transformer $sp_{\mathsf{cart}}(Q)$ is defined as $\alpha_{\mathsf{cart}}(sp(\mathtt{stmt}, \gamma_{\mathsf{bool}}(Q)))$. In order to compute the abstract transition relation we need to check for each predicate P_i ($\neg P_i$, respectively) and each minterm M_j whether $sp(\mathtt{stmt}, \gamma_{\mathsf{bool}}(M_j)))$ implies P_i, i.e., whether $\{\gamma_{\mathsf{bool}}(M_j)\}$ $\mathtt{stmt}$ $\{P_i\}$ is a valid Hoare triple. Since $\{\gamma_{\mathsf{bool}}(M_j)\}$ $\mathtt{stmt}$ $\{P_i\}$ can be stated as $\gamma_{\mathsf{bool}}(M_j) \Rightarrow wlp(\mathtt{stmt}, P_i)$, this amounts to finding all minterms M_j such that $\gamma_{\mathsf{bool}}(M_j)$ implies $wlp(\mathtt{stmt}, P_i)$. This process can be improved significantly by considering sub-conjuncts of minterms. Let $M_j = \bigwedge_{i=1}^{n} L_i$, $L_i \in \{\mathtt{x}_i, \neg\mathtt{x}_i\}$ and C be the conjunct $\bigwedge_{i \in I} L_i$, where $I \subseteq \{1, \ldots, n\}$. Since $M_i \Rightarrow C$, $\gamma_{\mathsf{bool}}(M_j)$ implies $wlp(\mathtt{stmt}, P_i)$ if $\gamma_{\mathsf{bool}}(C)$ does. The fact that conjuncts are partially ordered under implication enables a systematic search for the weakest set of conjuncts "covering" all minterms.

Example 2.4.4. *We derive the abstract transition relation presented in Example 2.4.3 by means of Cartesian abstraction. We compute*

$$wlp([\mathtt{i} \le 10], \mathtt{i} < 100) = (\mathtt{i} \le 10) \Rightarrow (\mathtt{i} < 100) = \textsf{\textit{true}} \qquad \textit{and}$$

$$wlp([\mathtt{i} \le 10], \mathtt{i} \ge 100) = (\mathtt{i} \le 10) \Rightarrow (\mathtt{i} \ge 100) = \textsf{\textit{false}}\,.$$

Therefore, we conclude that $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathsf{true})) \Rightarrow (\mathtt{i} < 100)$ *and the negated predicate* $(\mathtt{i} \geq 100)$ *is implied by* $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathsf{false}))$. *Thus,* $sp_{\mathsf{bool}}([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathsf{true})) \Rightarrow \gamma_{\mathsf{bool}}(\mathtt{x}_2)$ *and* $sp_{\mathsf{bool}}([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathsf{false})) \Rightarrow \gamma_{\mathsf{bool}}(\neg\mathtt{x}_2)$, *and from Definition 2.4.6 we obtain* $(\mathsf{true} \wedge \acute{\mathtt{x}}) \vee (\mathsf{false} \wedge (\neg\acute{\mathtt{x}}))$ *as transition relation for* $\mathtt{x}_2$.

Similarly, we derive

$$wlp([\mathtt{i} \leq 10], \mathtt{i} = \mathtt{j}) = (\mathtt{i} \leq 10) \Rightarrow (\mathtt{i} = \mathtt{j}) \qquad \textit{and}$$
$$wlp([\mathtt{i} \leq 10], \mathtt{i} \neq \mathtt{j}) = (\mathtt{i} \leq 10) \Rightarrow (\mathtt{i} \neq \mathtt{j})$$

and observe that $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathtt{x}_1)) \Rightarrow (\mathtt{i} = \mathtt{j})$ *and* $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\neg\mathtt{x}_1)) \Rightarrow (\mathtt{i} \neq \mathtt{j})$. *Note that there is no need to consider stronger conjuncts (for instance* $\mathtt{x}_1 \wedge \mathtt{x}_2$*). We obtain the constraints* $\acute{\mathtt{x}}_1 \wedge \grave{\mathtt{x}}_1$ *and* $(\neg\acute{\mathtt{x}}_1) \wedge (\neg\grave{\mathtt{x}}_1)$ *from* $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\mathtt{x}_1)) \Rightarrow \gamma_{\mathsf{bool}}(\mathtt{x}_1)$ *and* $sp([\mathtt{i} \leq 10], \gamma_{\mathsf{bool}}(\neg\mathtt{x}_1)) \Rightarrow \gamma_{\mathsf{bool}}(\neg\mathtt{x}_1)$, *respectively.*

Since the weakest liberal precondition distributes over conjunctions (see Lemma B.1.1 in Section B.1), we obtain the abstract transition relation by pairing the transition contraints for the separate Boolean variables into a single relation

$$\acute{\mathtt{x}}_2 \wedge ((\acute{\mathtt{x}}_1 \wedge \grave{\mathtt{x}}_1) \vee ((\neg\acute{\mathtt{x}}_1) \wedge (\neg\grave{\mathtt{x}}_1))),$$

which simplifies to $\acute{\mathtt{x}}_2 \wedge (\acute{\mathtt{x}}_1 \Leftrightarrow \grave{\mathtt{x}}_1)$.

Note that in comparison to Example 2.4.4 we lose some precision: Unlike the transition relation $(\grave{\mathtt{x}}_1 \Leftrightarrow \acute{\mathtt{x}}_1) \wedge \grave{\mathtt{x}}_2 \wedge \acute{\mathtt{x}}_2$, the relation $\acute{\mathtt{x}}_2 \wedge (\acute{\mathtt{x}}_1 \Leftrightarrow \grave{\mathtt{x}}_1)$ does not rule out that $\grave{\mathtt{x}}_2 = \mathsf{false}$. The reason for this imprecision is that we considered the preconditions for $\mathtt{x}_1$ and $\neg\mathtt{x}_1$, but not for the combination $\mathtt{x}_1 \wedge (\neg\mathtt{x}_1)$, which is also a possible outcome of α_{cart}. This can be amended by adding the constraint $\grave{\mathtt{x}}_2$, which derives from

$$wlp([\mathtt{i} \leq 10], \mathsf{false}) = \neg(\mathtt{i} < 10).$$

Similarly, the imprecision introduced by Cartesian approximations may result in the elimination of the correlation between two Boolean variables $\mathtt{x}_i$ and $\mathtt{x}_j$ (the corresponding predicates, respectively). This approximation can be refined by constraining the transition

relation on demand. A detailed exposition of this approach can be found in [BCDR04].

2.4.3 Boolean Programs

In practice, the relation $\widehat{\texttt{stmt}} \subseteq (\mathcal{L}_{\mathbb{B}} \times \mathcal{L}_{\mathbb{B}})$ is often represented as an assignment or a conditional statement. For instance, the relation derived in Example 2.4.3 corresponds to the condition $\texttt{[x}_2\texttt{]}$. We can verify this claim by showing that

$$sp_{\mathsf{bool}}([\mathtt{x}_2], \grave{\mathtt{x}}_1 = \mathtt{x}_1 \wedge \grave{\mathtt{x}}_2 = \mathtt{x}_2)[\mathtt{x}_1/\acute{\mathtt{x}}_1][\mathtt{x}_1/\acute{\mathtt{x}}_1] \quad = \quad (\grave{\mathtt{x}}_1 \Leftrightarrow \acute{\mathtt{x}}_1) \wedge \grave{\mathtt{x}}_2 \wedge \acute{\mathtt{x}}_2$$

holds (cf. page 28). The program statements presented in Section 2.1, however, are not general enough to represent all conceivable transition relations. Since the support predicates corresponding to the Boolean variables $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ track relations over variables in the original program, an assignment in the concrete program may trigger a change of more than one Boolean variable. The introduction of a parallel assignment $\mathtt{x_1}, \ldots, \mathtt{x_n} := e_1, \ldots, e_n$ resolves this issue. The semantics for this statement is provided in Table 2.2. In addition, we use `*` to represent a non-deterministic value, which may take the value of either true or false. The semantics of a non-deterministic assignment is determined by

$$sp_{\mathsf{bool}}(\mathtt{x} := *, P) = \exists v . \mathtt{x} = v \wedge P[\mathtt{x}/v]\,,$$

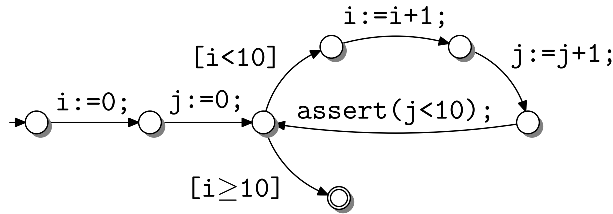
where v is a fresh variable not occurring in e and P. Programs comprising statements of this kind are called *Boolean programs*. They resemble the programs in Section 2.3 in structure, but allow only variables of Boolean type. A detailed discussion of this formalism exceeds the scope of this dissertation. The formal semantics of Boolean programs is provided in [BR00a]. Furthermore, we do not provide an algorithms to derive Boolean programs from abstract transition relations. Instead, we construct the statements in an *ad hoc* manner and refer the reader to [BMMR01] for details.

The following example demonstrates how abstractions can be represented as Boolean programs.

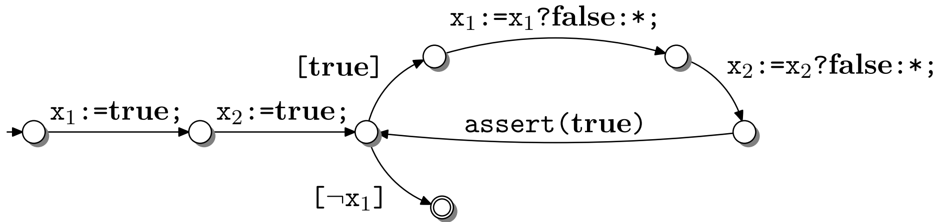
Example 2.4.5. *Consider the program in Figure 2.5(a). The Figure 2.5(b) shows the corre-*

Statement stmt	**Strongest Postcondition** $sp_{\mathsf{bool}}(\mathtt{stmt}, P)$
$\mathtt{x}_1,\ldots,\mathtt{x}_n\mathtt{:=}e_1,\ldots,e_n$	$\exists \mathtt{x}'_1,\ldots,\mathtt{x}'_n\,.\,(\mathtt{x}_1 = e_1[\mathtt{x}_1/\mathtt{x}'_1]\ldots[\mathtt{x}_n/\mathtt{x}'_n]) \wedge \ldots \wedge (\mathtt{x}_n = e_n[\mathtt{x}_1/\mathtt{x}'_1]\ldots[\mathtt{x}_n/\mathtt{x}'_n]) \wedge P[\mathtt{x}_1/\mathtt{x}'_1]\ldots[\mathtt{x}_n/\mathtt{x}'_n]$
$[R]$	$P \wedge R$
$\mathtt{assert}(R)$	$P \wedge R$
$\mathtt{stmt}_1;\mathtt{stmt}_2$	$sp_{\mathsf{bool}}(\mathtt{stmt}_2, sp_{\mathsf{bool}}(\mathtt{stmt}_1, P))$

Table 2.2: Semantics for Boolean program statements



(a) A CFA representing a program with a loop.



(b) A Boolean program representing an abstraction of the CFA above.

Figure 2.5: A program and its corresponding abstraction for the predicates $\mathcal{P} = \{(\mathtt{i}{=}0), (\mathtt{j}{=}0)\}$ represented by the Boolean variables $\mathtt{x}_1$ and $\mathtt{x}_2$, respectively

sponding Boolean program constructed using the predicates ($\mathtt{i}$*=0) and (*$\mathtt{j}$*=0) (represented by* $\mathtt{x}_1$ *and* $\mathtt{x}_2$*, respectively). The expression* $e_1?e_2{:}e_3$ *is shorthand for* $(e_1 \Rightarrow e_2) \wedge ((\neg e_1) \Rightarrow e_3)$*. For instance, the assignment* $\mathtt{x}_2 := \mathtt{x}_2?\textit{false}{:}\ *$ *which is the abstraction of* $\mathtt{j} := \mathtt{j} + 1$ *corresponds to the transition relation* $(\grave{\mathtt{x}}_2 \Rightarrow (\neg\acute{\mathtt{x}}_2)) \wedge (\grave{\mathtt{x}}_1 \Leftrightarrow \acute{\mathtt{x}}_1)$*. Accordingly, the approximation is not precise enough to show that the assertion* $(\mathtt{j} < 10)$ *(implied by* $\gamma_{\mathit{bool}}(\mathtt{x}_2)$*) in the original program holds.*

The previous example suggests that there is a direct correspondence between reachability trees for the predicate transformer $sp_{\mathsf{bool}} : \mathcal{L}_{\mathbb{B}} \to \mathcal{L}_{\mathbb{B}}$ and abstract reachability trees (Definition 2.4.3). The concretisation function γ_{bool} maps each set of reachable abstract states represented by $P \in \mathcal{L}_{\mathbb{B}}$ to an approximate predicate $\gamma_{\mathsf{bool}}(P) \in \widehat{\mathcal{L}}_{\mathcal{P}}$. Due to the finitary nature of transition systems represented by a CFA annotated with monotonic transition relations $\widehat{\mathtt{stmt}}$ we can apply existing (symbolic) model checking algorithms to compute the $MOP(sp_{\mathsf{bool}}, \mathsf{true})$. Instances of such algorithms are presented in [BR00b, Sch02] (for pushdown systems) and [McM93] (for finite state transition systems). Furthermore, the author and his collaborators presented SAT-based algorithms for the reachability analysis in Boolean programs in [BKW07a] and [BKW07b]. We do not cover these algorithms in the scope of this dissertation and refer the reader to these publications for more detail and related work.

2.5 Construction of Safety Proofs

In this section, we present a technique to generate Hoare proofs establishing the safety of programs. Intuitively, this is possible by expanding the abstract reachability tree of a program until we find approximate predicates representing inductive invariants for the cycles in the corresponding CFA. Based on this observation, we formalise the notion of a safety proof. In Example 2.4.2, however, we observe that an approximation may be too coarse to establish the safety of a program. In order to obtain safety proofs, it may therefore be necessary to improve the approximation. We discuss how spurious counterexamples can be used as a catalyst for such a refinement.

2.5.1 Complete Reachability Trees and Inductive Invariants

The definition of reachability trees (Definition 2.3.2) in Section 2.3 is driven by the system of equations determined by the statements of the CFA and the monotonic predicate transformer *sp*. Intuitively, the construction of a reachability tree for the predicate transformer *sp* corresponds to the fixed point iteration of the Equations (2.4) in the execution order dictated by the CFA.

Definition 2.5.1 (Complete Reachability Tree [Jha04, McM06])**.** *Let $T = (N_T, E_T)$ be a reachability tree of a CFA (N, E) for a predicate transformer $\widehat{pt}$ which is a correct upper approximation of the strongest postcondition. T is complete if the following holds for each node $(t, i) \in N_T$ with $i \in N$:*

1. *Either (t, i) is fully expanded, or*
2. *(t, i) is a leaf node and is* covered *by previously explored nodes, i.e.,*

$$\exists (t_1, i), \ldots, (t_m, i) \in N_T \, . \, \bigwedge_{j=1}^{m} (t_j < t) \wedge \left(P_{(t,i)} \Rightarrow \bigvee_{j=1}^{m} P_{(t_j,i)} \right) \tag{2.12}$$

   holds.

Lemma 2.5.1 (Reachility Trees, Inductive Invariants)**.** *Let $T = (N_T, E_T)$ be a complete reachability tree of the CFA (N, E) and a correct upper approximation $\widehat{pt}$ of sp. Then*

$$P_i = \bigvee_{(t,i) \in N_T} P_{(t,i)}$$

is an inductive invariant for the node $i \in N$ with respect to the strongest postcondition sp.

Proof. Assume that there is a P_i which is not an inductive invariant of the node $i \in N$. Then there exists a cyclic path $\pi \in (N, E)$ starting and ending in i such that $sp(\pi, P_i) \not\Rightarrow P_i$. Since sp distributes over disjunctions (Lemma B.1.1, Section B.1), we obtain $sp(\pi, P_i) = \bigvee_{(t,i) \in N_T} sp(\pi, P_{(t,i)})$, and therefore it must hold that $\exists (t, i) \in N_T \, . \, sp(\pi, P_{(t,i)}) \not\Rightarrow P_i$.

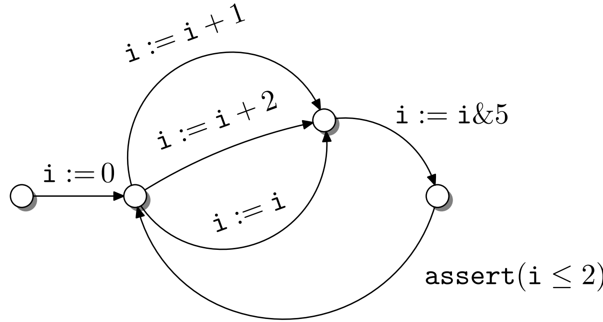
In order to show that such a node cannot exist, we prove a more general result. Given a node $(t, i) \in N_T$, we show that for any arbitrary path π starting at node i

and incident upon node j ($i, j \in N$), there exist nodes $(t_1, j), \ldots, (t_m, j) \in N_T$ such that $sp(\pi, P_{(t,i)}) \Rightarrow \bigvee_{l=1}^{m} P_{(t_l,j)}$. A similar proof can be found in [Jha04], Section 3.2. We prove our claim by induction on the length of π. If $\pi = \varepsilon$, then $sp(\pi, P_{(t,i)}) = P_{(t,i)}$, which establishes the base case. Consider the path π; `stmt` and suppose the induction hypothesis holds for π. We are done if the node reached by π is not a leaf node of T. Otherwise, let $(t_1, j), \ldots, (t_m, j)$ be nodes in N_T such that $sp(\pi, P_{(t,i)}) \Rightarrow \bigvee_{l=1}^{m} P_{(t_l,j)}$. Note that we can always choose these nodes in such a manner that $(t_1, j), \ldots, (t_m, j)$ are fully expanded nodes of T, for if one node (t_l, j) is an unexpanded leaf node, we can replace it with the nodes covering it. The recursive application of this replacement policy must eventually terminate, since the time-stamps of the replacement nodes are smaller than t_l. Since the predicate transformer sp is monotone in its second argument, it holds that $sp(\texttt{stmt}, sp(\pi, P_{(t,i)})) \Rightarrow sp(\texttt{stmt}, \bigvee_{l=1}^{m} P_{(t_l,j)})$. Moreover, since sp distributes over disjunctions, we obtain $sp(\texttt{stmt}, sp(\pi, P_{(t,i)})) \Rightarrow \bigvee_{l=1}^{m} sp(\texttt{stmt}, P_{(t_l,j)})$. Since all nodes (t_l, j) (where $1 \leq l \leq m$) are fully expanded, there must be m successor nodes in T labelled $sp(\texttt{stmt}, P_{(t_l,j)})$ ($1 \leq l \leq m$), respectively. (If there is a node (t_l, j) which has no such successor, then the path π; `stmt` does not exist in (N, E).) This concludes the argument. □

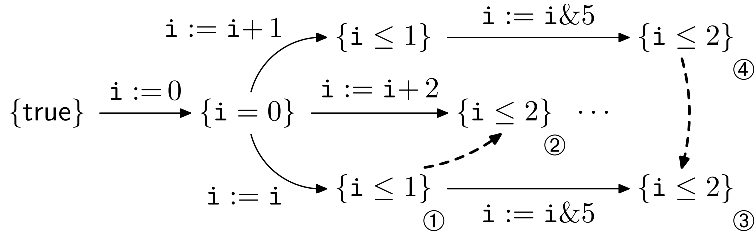
Reachability and Non-Monotonic Predicate Transformers

Definition 2.5.1 requires that all covered nodes are leaf nodes. At first glance, this may seem overly restrictive, since the expansion of a leaf node (n, i) might yield successor nodes which in turn may cover other nodes in the tree. The following example demonstrates why such an *indirect* coverage is problematic.

Example 2.5.1. *Consider the program in Figure 2.6(a). By simulating the program using the strongest postcondition we can convince ourselves that the program is safe and that its state space is finite. Figure 2.6(b) illustrates an attempt to construct a reachability tree for this program using a non-monotonic predicate transformer* $\widehat{pt}$. *The dashed arrows indicate coverage relations between nodes. Let us draw our attention to node ① which is labelled* $\texttt{i} \leq 1$ *and covered by node ② labelled* $\texttt{i} \leq 2$. *A further expansion of ① yields a node ③ labelled with the predicate* $\texttt{i} \leq 2$, *which covers the node ④. We could now be tempted to assume that we*



(a) A peculiar program



(b) An attempt to construct a safety proof using a non-monotonic predicate transformer

Figure 2.6: Non-monotonic reasoning and reachability

need not expand ③ and ④ further, since any successor of ① should be covered by successors of ②. This assumption, however, does not hold for non-monotonic predicate transformers. Expanding ② may yield the predicate $\widehat{pt}(\mathtt{i} := \mathtt{i}\&5, \mathtt{i} \leq 2) = (\mathtt{i} \leq 1)$*, which is too strong to cover the node ④. Therefore, the node ③ must not be used to cover node ④.*

A more generic example exposing this issue can be found in [McM06]. The fact that non-monotonic predicate transformers rule out indirect coverage imposes a restriction on the search algorithm used to construct safety proofs.

Example 2.5.2. *We continue working in the setting of Example 2.5.1. Consider a search algorithm which discovers that node ③ covers node ④ before it explores the branch leading to node ②. If node ② is later used to cover node ①, the algorithm needs to remove the covering relation between node ③ and node ④.*

Accordingly, covering one node may result in uncovering other nodes. A non-deterministic approach to compute a set of covering nodes may result in circular dependencies [McM06]. The restriction that a node can only be covered by a node with a smaller time-stamp warrants that covering a node (t_i, i) can only result in uncovering nodes (t_j, j) with $t_j > t_i$.

2.5.2 Inductive Invariants and Safety Proofs

Complete reachability trees can be either obtained by means of standard breadth-first or depth-first search algorithms or by means of model checking. Algorithms of the former nature are presented in [HJM+02, McM06]. In Section 2.4.3 we briefly touch upon model checking algorithms for Boolean programs, which enable the construction of complete reachability trees for the predicate transformer sp_{bool}. The current section is concerned with the extraction of safety proofs from complete reachability trees.

A complete reachability tree $T = (N_T, E_T)$ is sufficient to establish the safety of the corresponding CFA (N, E) if and only if for each node $i \in N$ followed by an assertion, the labels of all corresponding nodes in N_T are strong enough to entail that the assertion holds. Formally,

$$\forall \langle i,j\rangle \in E\,.\,(\mathtt{stmt}_{\langle i,j\rangle} = \mathtt{assert}(R)) \Rightarrow \big(\forall (t,l) \in N_T\,.\, l \neq i \vee (P_{(t,l)} \Rightarrow R)\big)\,. \tag{2.13}$$

Given such a complete reachability tree, Lemma 2.5.1 enables the construction of a safety proof.

Definition 2.5.2 (Safety Proof)**.** *A safety proof for a CFA (N, E) is a total mapping from the nodes in $i \in N$ to predicates in $P_i \in \mathcal{L}$ such that for each $\langle i,j\rangle \in E$ the Hoare triple $\{P_i\}$ $\mathtt{stmt}_{\langle i,j\rangle}$ $\{P_j\}$ holds and for each node $i \in N$ and all its outgoing edges labelled* $\mathtt{assert}(R)$*, it holds that $P_i \Rightarrow R$.*

The inductive invariants determined by Lemma 2.5.1 and a complete reachability tree satisfying (2.13) constitute such a safety proof. (The correctness of this claim follows immediately from the proof of Lemma 2.5.1.) The following example illustrates the close relationship between complete reachability trees and safety proofs.

Example 2.5.3. *Figure 2.7 shows a complete reachability tree for the CFA in Figure 1.1. This reachability tree is obtained using the predicate transformer determined by the interpolating decision procedure outlined in Section 1.3.1. We provide a detailed description of this decision procedure in Chapter 3, Section 3.5.*

In the following, we provide a Hoare triple $\{P\}$ $\mathtt{stmt}_{\langle i,j\rangle}$ $\{Q\}$ for each edge $\langle (t_1, i), (t_2, j)\rangle$

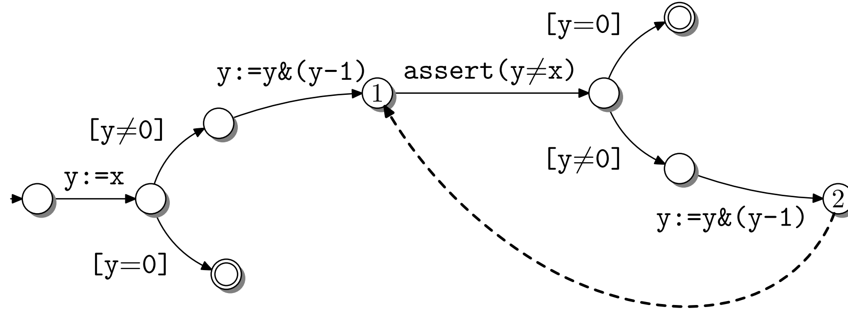


Figure 2.7: A complete reachability tree for the program in Figure 1.1. The corresponding annotations are listed in Example 2.5.3.

of the reachability tree in Figure 2.7. A proof of the validity of each Hoare triple is provided in Section 1.3.1.

$$\{\textit{true}\}\ \mathtt{y} := \mathtt{x}\ \{\mathtt{y} = \mathtt{x}\}$$

$$\{\mathtt{y} = \mathtt{x}\}\ [\mathtt{y} \neq 0]\ \{\mathtt{y} = \mathtt{x} \wedge \mathtt{y} \neq 0\}$$

$$\{\mathtt{y} = \mathtt{x} \wedge \mathtt{y} \neq 0\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y} - \mathtt{1})\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ ①$$

$$\{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ \mathtt{assert}(\mathtt{x} \neq \mathtt{y})\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}$$

$$\{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ [\mathtt{y} \neq 0]\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0) \wedge (\mathtt{y} \neq 0)\}$$

$$\{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0) \wedge (\mathtt{y} \neq 0)\}\ \mathtt{y} := \mathtt{y}\&(\mathtt{y} - \mathtt{1})\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ ②$$

The annotations required for a safety proof can be easily obtained from these Hoare triples. The predicate transformer labels the nodes ① and ② with $\mathtt{y} \leq (\mathtt{x} - 1) \wedge (\mathtt{x} \neq 0)$*. Therefore, node ② is covered by ① (indicated by the dashed arrow). All other nodes in the reachability tree are fully expanded. It follows from Lemma 2.5.1 that* $\mathtt{y} \leq (\mathtt{x} - 1) \wedge (\mathtt{x} \neq 0)$ *is an inductive invariant for the respective node in the CFA in Figure 1.1. We obtain the Hoare triples*

$$\{\textit{true}\}\ \mathtt{y} := \mathtt{x}; [\mathtt{y} \neq 0];\ \mathtt{y} := \mathtt{y}\&(\mathtt{y} - \mathtt{1})\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ ①$$

$$\{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ \mathtt{assert}(\mathtt{x} \neq \mathtt{y}); [\mathtt{y} \neq 0];\ \mathtt{y} := \mathtt{y}\&(\mathtt{y} - \mathtt{1})\ \{\mathtt{y} \leq (\mathtt{x} - \mathtt{1}) \wedge (\mathtt{x} \neq 0)\}\ ②$$

which establish that $\mathtt{y} \leq (\mathtt{x} - 1) \wedge (\mathtt{x} \neq 0)$ *is an inductive invariant.*

Remarks. The complete reachability tree in Example 2.5.3 corresponds to a safe, complete, and well-labelled unwinding (as presented in [McM06]). Reachability trees with cut-points correspond to the tagged graphs presented in [Kin70].

2.5.3 Refining Approximations

Example 2.4.2 demonstrates that abstract reachability trees may give rise to spurious counterexamples (Definition 2.4.4). While an approximation of finitary nature such as predicate abstraction always enables us to construct a complete reachability tree, this tree may not satisfy the condition (2.13). If this is the case, it is possible to improve the accuracy of the approximation by means of counterexample-guided abstraction refinement (CEGAR) [BSV93, Kur94, CGJ+00]. The aim of this technique is to eliminate a spurious counterexample from an abstract reachability tree (Definition 2.4.3). This can be achieved by improving the accuracy of the approximate predicate transformer $\widehat{pt}$. In Sections 2.2 and 2.4.1 we encounter two types of approximate predicate transformers: An interpolation-based parametrised predicate transformer $\overrightarrow{pt}(\texttt{stmt}, Q, P)$ and the predicate-abstraction based approximation $\widehat{sp}$. Accordingly, we present an appropriate refinement strategy for each of these approximations. Let π be a path representing a spurious counterexample in an abstract reachability tree T reaching an assertion `assert`(R).

- If T is derived from a parametrised predicate transformer $\overrightarrow{pt}(\texttt{stmt}, Q, P)$ as defined at the end of Section 2.2, we tighten the upper bounds Q for the nodes visited by π. For each such node $(t, i) \in T$, the new upper bound is determined by $wlp(\pi', R)$, where π' represents the suffix of π starting at i.
- If T is the result of a predicate-abstraction-based approximation, we compute a sequence of predicates $P_1, \ldots, P_n$ satisfying the chain condition (Definition 2.2.1) and strong enough to establish $\{\textsf{true}\}\ \pi\ \{R\}$. Then, we add these predicates to $\mathcal{P}$ (and $\widehat{\mathcal{L}}_{\mathcal{P}}$, respectively).

The first approach is applied in [McM06], and variations of the latter approach are presented in [BR02a, HJMS02, HJMM04]. We provide a brief outline of these techniques.

For an overview of predicate abstraction-based refinement techniques we refer the reader to our survey [DKW08].

Refining Interpolation-based Approximations

McMillan's interpolation-based verification technique (presented in [McM06]) iteratively approximates the upper bound Q for a parametrised predicate transformer $\overrightarrow{pt}(\pi, Q, P)$. Let (N, E) be a CFA. We assume, w.l.o.g., that the CFA contains only a single assertion $\texttt{assert}(\mathrm{R})$, and that this assertion is the annotation an outgoing edge of $j \in N$. Analogously to the merge over all paths (Definition 2.3.1), we derive the weakest sufficient upper bound for each node $i \in N$ as

$$Q_i = \bigwedge_{\pi_i \in \Pi(i,j)} wlp(\pi_i, R)\,, \tag{2.14}$$

where $\Pi(i, j)$ denotes the set of all paths from i to j in the CFA. If we combine Equation (2.14) and Definition 2.3.1, we obtain

$$P_i = \bigvee_{\pi_i \in \Pi(\to\!\bigcirc, i)} \overrightarrow{pt}\left(\pi_i, \left(\bigwedge_{\pi_j \in \Pi(i,j)} wlp(\pi_j, R)\right), \mathsf{true}\right) \tag{2.15}$$

as the $MOP(\overrightarrow{pt}, \mathsf{true})$ for each node $i \in N$ that is sufficiently tight to establish the safety of the program.

A spurious counterexample in a reachability tree T constitutes a violation of the upper bound (2.14). Let $\pi = \texttt{stmt}_1; \ldots ; \texttt{stmt}_n$ represent a spurious counterexample violating an assertion $\texttt{assert}(R)$ and let $Q_{(t_1,1)}, \ldots, Q_{(t_n,n)}$ represent the corresponding upper bounds for the nodes $(t_1, 1), \ldots, (t_n, n)$ of the corresponding path in the reachability tree T. Then we obtain a set of predicates inductively defined by

$$Q'_{(t_n,n)} = R\,, \quad Q'_{(t_i,i)} = wlp(\texttt{stmt}_i, Q'_{(t_{(i+1)},(i+1))}) \text{ for } 1 \leq i < n\,.$$

These predicates are sufficient to construct a Hoare proof for $\{\mathsf{true}\}\ \pi\ \{R\}$. The predicates $Q_{(t_i,i)} \wedge Q'_{(t_i,i)}$ (for $1 \leq i \leq n$) constitute the new upper bounds for $(t_1, 1), \ldots, (t_n, n)$, effectively eliminating the spurious counterexample from any reachability tree computed

using the refined predicate transformer $\overrightarrow{pt}(\pi_i, Q_{(t_i,i)} \wedge Q'_{(t_i,i)}, \textsf{true})$.

Example 2.5.4. *Consider the reachability tree in Figure 2.7. Suppose that, initially, the upper bounds for all nodes are* $\textsf{true}$. *Accordingly, the reachability tree constructed using the approximate predicate transformer* $\overrightarrow{pt}(\texttt{stmt}, \textsf{true}, P)$ *contains a counterexample* $\texttt{y} := \texttt{x}$; $[\texttt{y} \neq \texttt{0}]$; $y := y\&(y-1)$ *violating the assertion* $\texttt{assert}(\texttt{x} \neq \texttt{y})$. *By refining the initial upper bounds to*

$$
\begin{aligned}
&(\texttt{x} \neq \texttt{y})\,, \\
&\texttt{x} \neq \texttt{y}\&(\texttt{y}-\texttt{1}) \;=\; wlp(\texttt{y} := \texttt{y}\&(\texttt{y}-\texttt{1}), (\texttt{x} \neq \texttt{y}))\,, \quad \textit{and} \\
&(\texttt{y} = \texttt{0}) \vee (\texttt{x} \neq \texttt{y}\&(\texttt{y}-\texttt{1})) \;=\; wlp([\texttt{y} \neq \texttt{0}], (\texttt{x} \neq \texttt{y}\&(\texttt{y}-\texttt{1})))\,,
\end{aligned}
$$

respectively, we obtain (according to Table 1.2) the Craig interpolants

$$
\begin{aligned}
\texttt{y} \leq \texttt{x} \;&=\; \overrightarrow{pt}(\texttt{y} := \texttt{x},\ (\texttt{y} = \texttt{0}) \vee (\texttt{x} \neq \texttt{y}\&(\texttt{y}-\texttt{1})),\ \textsf{true})\,, \\
\texttt{y} \leq \texttt{x} \wedge \texttt{y} \neq \texttt{0} \;&=\; \overrightarrow{pt}([\texttt{y} \neq \texttt{0}],\ \texttt{x} \neq \texttt{y}\&(\texttt{y}-\texttt{1}),\ \texttt{y} \leq \texttt{x}) \quad \textit{and} \\
\texttt{x} \neq \texttt{0} \wedge \texttt{y} \leq (\texttt{x}-\texttt{1}) \;&=\; \overrightarrow{pt}(\texttt{y} := \texttt{y}\&(\texttt{y}-\texttt{1}),\ \texttt{x} \neq \texttt{y},\ \texttt{y} \leq \texttt{x} \wedge \texttt{y} \neq \texttt{0})\,.
\end{aligned}
$$

The inductive invariant $\texttt{x} \neq \texttt{0} \wedge \texttt{y} \leq (\texttt{x}-\texttt{1})$ *is sufficiently strong to rule out the violation of the assertion. Thus, the refinement presented above eliminates the spurious counterexample.*

Eager versus Lazy Refinement. According to Equation (2.14), an improved upper bound for a node $(t, i) \in N_T$ is also a valid upper bound for any other node $(t', i) \in N_T$. It is therefore admissible to refine all these nodes accordingly. The approach refining only the (t, i) along the path in N_T which corresponds to the counterexample is known as lazy refinement and was first presented in [HJMS02]. The approach refining *all* nodes in N_T which correspond to the CFA node i is known as eager refinement and used in [BR02a, CKSY05], for instance.

The refinement of an approximation makes it necessary to recompute certain sections of the abstract reachability tree $T = (N_T, E_T)$. Furthermore, this reconstruction of T may invalidate previously established coverage relations in T. The refinement, however,

potentially eliminates more than one spurious counterexample, avoiding a repeated reconstruction of the reachability tree. Intuitively, eager refinement eliminates a larger number of spurious counterexamples than lazy refinement, but requires the reconstruction of the entire reachability tree.

Refining Predicate Abstraction-based Approximations

The mechanism underlying the refinement of predicate abstraction-based approximations is the relaxation of the restrictions imposed by the predicate language $\widehat{\mathcal{L}}_{\mathcal{P}}$. Given an abstraction function $\alpha : \mathcal{L} \rightarrow \widehat{\mathcal{L}}_{\mathcal{P}}$, the set of predicates $\mathcal{P}$ defining $\widehat{\mathcal{L}}_{\mathcal{P}}$ determines the accuracy of the approximation. This accuracy can be improved by increasing the expressiveness of $\widehat{\mathcal{L}}_{\mathcal{P}}$ by adding additional predicates to $\mathcal{P}$.

Similarly to the refinement approach in the previous section, the goal is to refine $\widehat{\mathcal{L}}_{\mathcal{P}}$ until the weakest sufficient upper bound (2.14) is expressible for all nodes of the CFA. Again, the strategy is to iteratively eliminate spurious counterexamples.

Given a spurious counterexample $\texttt{stmt}_1; \ldots ; \texttt{stmt}_n$ violating the assertion $\texttt{assert}(R)$, it is possible to compute a sequence of predicates $P_1, \ldots, P_n$ (where $P_n = R$) satisfying the chain condition in Definition 2.2.1. Suppose we add these predicates to $\mathcal{P}$, thus increasing the expressiveness of $\widehat{\mathcal{L}}_{\mathcal{P}}$. By definition of the best correct upper approximation (Definition 2.4.2), this modification is sufficient to eliminate the spurious counterexample.

Example 2.5.5. *Consider the Boolean program in Figure 2.5. We demonstrate in Example 2.4.5 that the approximation constructed using the predicates (*$\texttt{i}=0$*) and (*$\texttt{j}=0$*) is not sufficient to establish the safety of the program in Figure 2.5(a). We obtain a spurious counterexample*

$$\texttt{x}_1 := \textit{true};\ \texttt{x}_2 := \textit{true};\ [\textit{true}];\ \texttt{x}_1 := \texttt{x}_1 ? \textit{false} : *;\ \texttt{x}_2 := \texttt{x}_2 ? \textit{false} : *$$

potentially violating the assertion $\texttt{j} < 10$*. The corresponding concrete path is*

$$\texttt{i} := 0;\ \texttt{j} := 0;\ [\texttt{i} < 10];\ \texttt{i} := \texttt{i} + 1;\ \texttt{j} := \texttt{j} + 1\,,$$

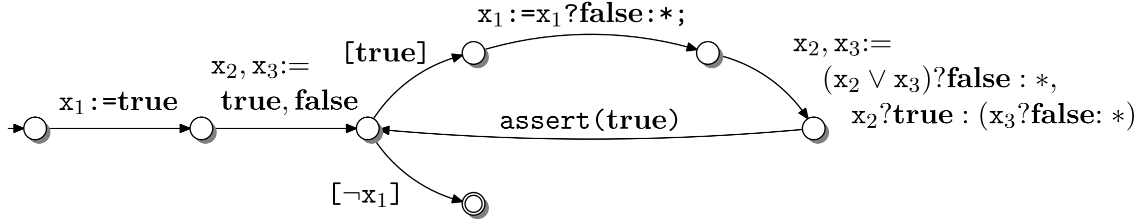


Figure 2.8: A refined approximation of the program in Figure 2.5(a) for the predicates (i=0), (j=0), and (j=1) (x_1, x_2, and x_3 respectively)

and by simulating this path using the strongest postcondition, we obtain the new predicates $i < 10$, $i = 1$, *and* $j = 1$ *and add them to* $\mathcal{P}$. *Figure 2.8 shows a refined version of the Boolean program in Figure 2.5(b). The predicates (i=0), (j=0), and (j=1) correspond to the Boolean variables* x_1, x_2, *and* x_3, *respectively. We omit the remaining predicates, since they are not necessary to eliminate the spurious counterexample.*

By constructing the abstract reachability tree for the refined Boolean program, we can verify that x_3 *holds after the first iteration of the loop and that the counterexample has indeed been eliminated.*

Boolean programs can be understood as an implicit representation of abstract reachability trees for the predicate transformer sp_{bool}. However, the fact that the same abstract transition function $\widehat{\texttt{stmt}}$ approximates all occurrences of `stmt` in the reachability tree effectively rules out lazy refinement. Therefore, all verification tools using Boolean programs as internal representation (such as Slam [BR02b, BCLR04] and SatAbs [CKSY05]) deploy eager refinement strategies.

2.5.4 Counterexample-Guided Abstraction Refinement

CEGAR is a framework comprising the steps described in Sections 2.4.1, 2.5.1, and 2.5.3 – abstraction, verification using reachability trees, and refinement. The term CEGAR denotes an iterative approach which, starting from a coarse approximation of the concrete program, refines the abstraction successively until the program can be proved either safe or faulty. Figure 2.9 illustrates this abstraction refinement cycle. Since this dissertation discusses different flavours of abstraction, reachability checks, and refinement, the flow-graph in Figure 2.9 is held as general as possible. It illustrates four phases of the process:

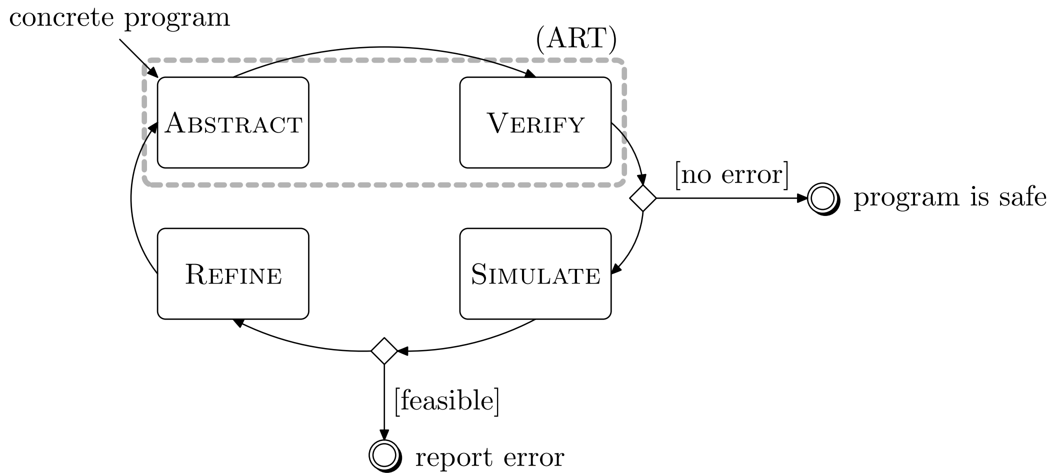


Figure 2.9: The Counterexample-Guided Abstraction Refinement scheme

1. ABSTRACT. The generation of Boolean programs is performed in this phase. In the case of a lazy refinement approach, abstraction and verification are performed in one single phase (indicated by the dashed box in Figure 2.9).

2. VERIFY. In the verification phase, one of the following tasks is performed:

   (a) verification of the Boolean program (generated in the previous phase) by means of model checking, a process which may yield a counterexample, or

   (b) expansion of the abstract reachability tree T until either an abstract counterexample is found or T is complete.

3. SIMULATE. The potentially spurious abstract counterexample obtained in the previous phase is simulated in the context of the concrete program. Legitimate counterexamples are reported.

4. REFINE. Spurious counterexamples are eliminated using the techniques described in Section 2.5.3.

The next example, which we owe to [JM06], demonstrates that the CEGAR loop does not necessarily terminate.

Example 2.5.6. *Figure 2.10 shows a program which is safe because it contains only one cyclic path* $[\mathtt{x} \neq 0]$; $\mathtt{x} := \mathtt{x} + 1$; $\mathtt{y} := \mathtt{y} + 1$ *which maintains the invariant* $(\mathtt{i} = \mathtt{j}) \Rightarrow (\mathtt{x} = \mathtt{y})$.

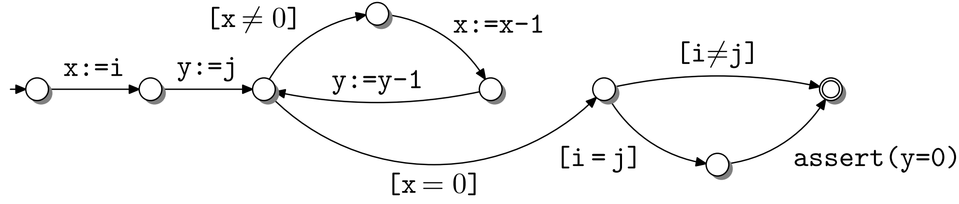


Figure 2.10: A program presented by Jhala and McMillan in [JM06]

Suppose we use predicate abstraction to generate an approximate version of the concrete program. We argue that, if we use the strongest postcondition to refine the approximation, we obtain the following spurious counterexample after n iterations of the CEGAR loop:

$$\mathtt{x} := \mathtt{i};\ \mathtt{y} := \mathtt{j};\ \Vert\!:\ [\mathtt{x} \neq 0];\ \mathtt{x} := \mathtt{x} - 1;\ \mathtt{y} := \mathtt{y} - 1;\ :\!\Vert_{n-1}\ [\mathtt{x} = 0];\ [\mathtt{y} = \mathtt{j}]\,,$$

(where the repetition sign denotes $(n-1)$ iterations of the cycle). This is easy to see for $n = 1$, assuming the initial approximation uses $\mathcal{P} = \{\mathit{true}, \mathit{false}\}$. A simulation of this path using the strongest postcondition yields the predicates $\{\mathtt{x} = \mathtt{j}, \mathtt{y} = \mathtt{j}, \mathtt{x} = 0, \mathtt{i} = \mathtt{j}\}$. The best approximation of

$$sp(\mathtt{x} := \mathtt{i};\ \mathtt{y} := \mathtt{j};\ [\mathtt{x} \neq 0];\ \mathtt{x} := \mathtt{x} - 1;\ \mathtt{y} := \mathtt{y} - 1, \mathit{true}) \ = \\ (\mathtt{x} - 1 = 0) \wedge (\mathtt{x} - \mathtt{1} = \mathtt{i}) \wedge (\mathtt{y} - \mathtt{1} = \mathtt{j})$$

*is therefore **true**, which is insufficient to prove the safety of the program. The subsequent iterations yield the refinement predicates $\{\mathtt{x} - (n-1) = \mathtt{j}, \mathtt{y} - (n-1) = \mathtt{j}\}$, resulting in the construction of a "one-hot" counter in the abstract domain. This process only terminates if the domain of the variables in the concrete program is finite (cf. Example 2.3.1).*

Due to Turing's undecidability result [Tur36], the fact that the CEGAR loop may fail to terminate is not surprising. However, the interesting aspect is that the non-termination in Example 2.5.6 can be avoided by choosing a different refinement strategy. Such strategies are discussed in Chapter 4 and in [JM06].

2.6 Verification Tools and Related Work

The SLAM toolkit [BR02b] (now commercialised under the name of *Static Driver Verifier* [BCLR04]), a CEGAR-based verification tool developed at Microsoft Research, popularised the use of predicate abstraction for software model checking. It was followed by BLAST [HJMS02], MAGIC [CCG+04], SATABS [CKSY05], and the F-SOFT verification framework [IYG+05], all of which are widely known CEGAR-based verification tools using predicate abstraction. A detailed comparison of these tools can be found in our survey [DKW08]. Recently, Microsoft research presented SLAM's reincarnation SLAM 2.0 and a derivative YOGI [NR10]. The latter is based on test generation.

The verification tool SLAM makes a clear distinction between the different phases of CEGAR (see Figure 2.9). It comprises three different tools, namely the predicate abstraction engine C2BP [BMMR01], a BDD-based model checker BEBOP [BR00b] for Boolean programs, and the simulation and refinement tool NEWTON [BR02a].

The structure underlying SATABS is similar. SATABS uses propositional logic and efficient satisfiability-solving techniques to implement the concrete predicate transformers and their respective approximations. This enables a bit-level accurate presentation of programs, making it possible to model bitwise operations and arithmetic overflow. The importance of a bit-level accurate representation of the program semantics is discussed at the end of Section 1.3.1 and in more detail in Chapter 3.

BLAST is the prototype of all tools based on lazy refinement. At the same time, it is the only predicate-abstraction based exponent in our list which uses lazy refinement. Recent versions of BLAST use interpolation to refine abstractions in a lazy manner [HJMM04]. The algorithm deployed in McMillan's verification tool IMPACT [McM06] resembles the lazy approach of BLAST but abandons predicate abstraction as approximation technique. Our verification tool WOLVERINE [KW09a] follows the same approach.

Chapter 2 omits the discussion of two prominent features of programming languages: procedures and concurrency. The support of these features is orthogonal to the techniques which are the focus of this dissertation. All the tools listed above are able to handle procedure calls, either based on abstracting the predicate transformers for procedure calls [Nel89]

(the approach taken by Slam [BMMR01]) or by inline expansion of procedures (as implemented in Impact, Wolverine, and earlier versions of SatAbs). Note that inline expansion is only a partial solution which does not support recursive function calls. Blast uses interpolation and introduces appropriate cut-points for paths in order to generate predicates over the arguments and return values of procedures [HJMM04]. This enables the summarisation of procedures [SP81]. A similar approach is proposed in [HHP10].

Furthermore, some of the verification tools discussed in this section provide limited or experimental support for concurrency. Magic deploys a compositional approach, decomposing the program into several components which are verified separately. Magic is able to verify programs using message passing, but does not support shared memory.

In the course of an internship at Microsoft Research, the author of this dissertation integrated support for concurrent programs into an experimental version of Slam. For this purpose, we replaced Bebop by the explicit state model checker Zing [AQRX04] and the SAT-based tool Boppo [CKS06], both of which enable the verification of concurrent Boolean programs. A similar modification to SatAbs is presented by the author and his collaborators in [WBKW07], where the benchmarks suggest that the verification of concurrent Boolean programs forms the bottleneck of this approach.

An adapted version of Blast supports threads by exploring the state space of one thread at a time, using CEGAR to approximate the behaviour of the thread and its environment [HJMQ03]. This modification enables the detection of race conditions in programs with shared memory.

The following two chapters focus on the extraction of Craig interpolants from counterexamples. The techniques in these two chapters are mutually orthogonal. Moreover, they are general enough to make an integration into existing implementations (such as the tools presented in this section) feasible.

Chapter 3

Constructing Interpolants

After the detailed discussion of applications of Craig's interpolation theorem in Chapter 2, we dedicate the current chapter to the construction of Craig interpolants. In particular, we focus on algorithms which extract interpolants from refutation proofs. While this is not the only approach to obtain interpolants ([RSS07], for instance, proposes to use a constraint solver to eliminate symbols local to a partition), this technique is widely and successfully applied by a number of tools [HJMM04, McM05, McM06, KW07, BZM08, KW09a, CGS10].

First, we provide a definition of the first-order language and the underlying theory we use to reason about software programs. Section 3.1 describes bit-vector logic, a formalism which reflects the finitary nature of the domain in which program variables take their values. We proceed with a general discussion of refutation proofs and their relation to interpolation (Section 3.2). The concepts introduced in Section 3.2 are then refined and applied to refutations of propositional encodings of bit-vector formulae in Section 3.3. We present a family of interpolation systems for refutations of propositional formulae, enabling us to derive a range of different interpolants. Section 3.4 discusses how proofs over propositional encodings can be lifted to a higher level of abstraction while still maintaining the advantages of a propositional representation. This technique enables us to compute proofs over "word-level" formulae. Interpolation techniques for such proofs are presented in Section 3.5. We discuss related work in Section 3.7 and present an experimental evaluation of our techniques in Section 3.6.

Table 3.1: The grammar for bit-vector arithmetic formulae [KS08]

formula	::=	*formula* $\wedge$ *formula* $\mid$ *formula* $\vee$ *formula* $\mid$ $\neg$*formula* $\mid$ *atom*
atom	::=	*propositional identifier* $\mid$ *term* $\rhd$ *term*
term	::=	$\sim$*term* $\mid$ *constant* $\mid$ *identifier* $\mid$ *term* $\bigcirc$ *term*

(where $\rhd \in \{=, \geq, >, \neq\}$ and $\bigcirc \in \{+, -, \cdot, \&, |, \oplus, \ll, \gg\}$)

Contribution. This chapter presents a number of novel interpolation techniques (published by the author and his collaborators in [KW07, KW09a] and [DPWK10]). The interpolation system presented in Section 3.3 and [DPWK10] is a generalisation of existing interpolation systems for propositional resolution proofs and relates the resulting interpolants by logical strength. The "proof lifting" technique introduced in Section 3.4 is a generalisation of the results presented in [KW07] and represents a combination of decision procedures which eagerly generate a propositional encoding of bit-vector formulae and *satisfiability modulo theory*-solvers (SMT-solvers). Finally, Section 3.5 presents two interpolation systems for bit-vector formulae and compares them with respect to the strength of the interpolants they generate. To the best of our knowledge, this is the first comparison of this kind for word-level interpolation systems.

3.1 Bit-Vector Arithmetic

Throughout Section 2.1, we refer to a first-order language $\mathcal{L}$ over a concrete domain of program states $\mathcal{C}$ without specifying any further details. The example in Section 1.3.1 illustrates that our application dictates specific requirements for $\mathcal{L}$. Program variables typically take values in a finite domain, and the semantics of the operators is determined by their respective implementation in hardware (see Figures 1.6 and 3.1, for instance). The theory of *bit-vector arithmetic* enables sound reasoning in such a setting. Therefore, we fix bit-vector logic as the language underlying the verification techniques presented in this dissertation. We adopt the notation and the semantics of Kroening and Strichman [KS08].

Syntax. Table 3.1 presents the abstract syntax of quantifier-free formulae in the theory of bit-vector arithmetic. We embed the quantifier-free language fragment specified by this grammar into the first-order language $\mathcal{L}$. We fix an enumerable set of variables, function

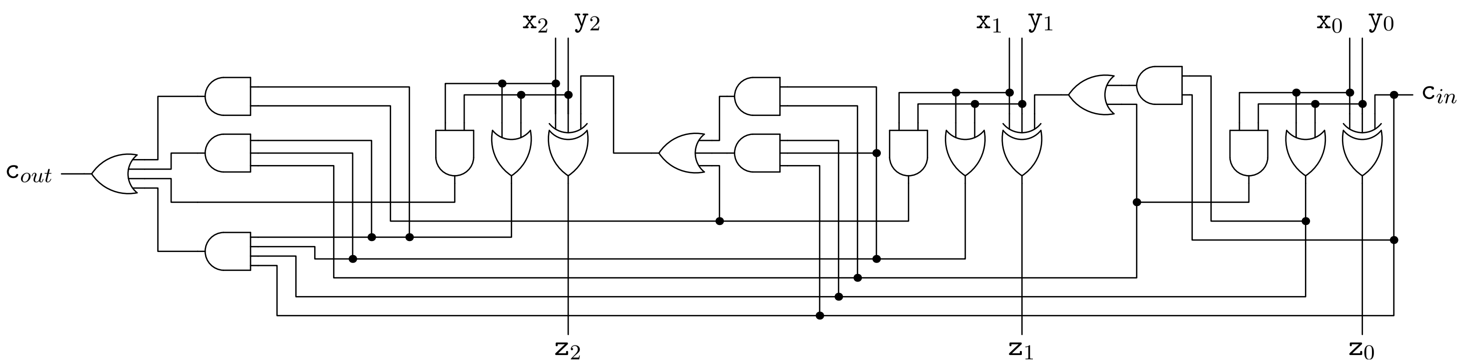


Figure 3.1: The semantics of 3-bit bit-vector addition

and predicate symbols, and constant symbols. Furthermore, we consider a subset of the variables to be propositional variables. Well-formed elements of $\mathcal{L}$ are generated by the following set of rules:

- A *term* t is a constant, a variable, or an application $f(t_1, \ldots t_n)$ of an n-ary function symbol f to terms $t_1, \ldots, t_n$. We fix a set of binary functions $\{+, -, \cdot, \&, |, \ll, \gg, \oplus\}$ which we allow to occur in infix notation. Moreover, the function $\sim$ is unary.
- An *atom* is a n-ary relation (or predicate) R applied to n terms $t_1, \ldots, t_n$. We fix a set of binary relations $\{=, \geq, >, \neq\}$ which may occur in infix notation (e.g., $t_1 \neq t_2$). Note that the set of atoms over the relations $\rhd \in \{=, \geq, >, \neq\}$ is closed under negation, i.e., the negation $\neg(t_1 \rhd t_2)$ of an atom can be expressed in terms of an atom.
- An non-quantified *formula* is a Boolean combination of atoms. We allow the Boolean operations $\wedge$, $\vee$, and $\neg$ and use $A \Rightarrow B$ to denote $\neg A \vee B$ and $A \Leftrightarrow B$ to denote $(A \Rightarrow B) \wedge (B \Rightarrow A)$. As usual, we assume that $\wedge$ has a higher precedence than $\vee$. Furthermore, given an atom A, we refer to A and $\neg A$ as *literals*.

We augment the language $\mathcal{L}$ with quantifiers in the standard way. Given a formula A in the first order language $\mathcal{L}$, let $\text{Var}(A)$ denote the non-quantified (free) variables in A. A formula is *closed* if it has no free variables. We use the symbols $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$, $\ldots$ to represent variables. In the context of Craig's interpolation theorem (Theorem 2.2.1), we treat these variables as constants (uninterpreted nullary functions, respectively) if they are not quantified, since the theorem applies to closed formulae exclusively (see Section 2.2).

Interpretations. A formula adhering to the syntactic rules introduced in the previous section may contain logical and non-logical symbols. Quantifiers, logical connectives, variables, and the equality relation are logical symbols, and so are parenthesis determining the precedence of operators in formulae. Furthermore, we include the truth constants true and false in the set of logical symbols and assume that the corresponding elements are contained in the domain $\mathcal{C}$. We use the standard interpretation of the logical connectives and the relation symbols $=$ and $\neq$. Universal and existential quantifiers range over the elements

of the domain $\mathcal{C}$ unless applied to propositional variables, in which case they range over the Boolean domain $\mathbb{B}$.

Non-logical symbols represent predicates (or relations), functions, and constants on the domain of discourse. A model $\mathcal{M}$ of a formula F comprises a domain $\mathcal{C}$ and an interpretation function assigning meaning to the non-logical symbols in F. We use the notation $f^{\mathcal{M}}$ to refer to the element in $\mathcal{C}$ assigned to the symbol f by the meaning function. Accordingly, an n-ary function symbol f (predicate symbol p) has a function $f^{\mathcal{M}} : \mathcal{C}^n \to \mathcal{C}$ ($p^{\mathcal{M}} : \mathcal{C}^n \to \mathbb{B}$, respectively) associated to it. A formula is satisfied by a model $\mathcal{M}$ if it evaluates to true in the respective interpretation. A formula is unsatisfiable if there exists no model satisfying it. A formula F is a tautology if no model satisfies its negation $\neg F$. A formula F entails a formula G, denoted as $F \models G$, if all models satisfying F also satisfy G.

We identify a fixed set of *interpreted* functions and predicates whose meaning is uniquely determined by the theory of bit-vectors. Conversely, *uninterpreted* function symbols and predicates have no other predetermined property than their name and arity. We use $\mathrm{Sym}(A)$ to denote the uninterpreted function and predicate symbols in a formula A.

In the following, we briefly outline the semantics of the interpreted functions and predicates of bit-vector arithmetic. The relation $\geq$ is a partial order over the domain $\mathcal{C}$, and $t_i > t_j$ denotes $(t_i \geq t_j) \wedge (t_i \neq t_j)$. The functions $+$ and $-$ correspond to addition and subtraction in modular arithmetic (see, for instance, [JCG09]). The modulus is determined by the cardinality of the finite domain the function result and parameters range over.

The finitary nature of the domain of program variables enables the representation of values $d \in \mathcal{C}$ as bit-vectors $d_{n-1} \dots d_0$, where $d_i \in \mathbb{B}$ (for $0 \leq i < n$) and n is the *width* of the bit-vector. Overloading our notation, we use 0 and 1 as an alternative representation for the constants false and true and interpret them accordingly in the context of arithmetic expressions. In the case of finite (bounded) domains containing unsigned or signed integers we assign each bit-vector a corresponding value in $\mathbb{N}_0$ and $\mathbb{Z}$, respectively:

$$(d_{n-1} \dots d_0)^{\mathcal{M}} \stackrel{\text{def}}{=} \begin{cases} \sum_{i=0}^{n-1} d_i \cdot 2^i & \text{for unsigned domains} \\ -2^{n-1} \cdot d_{n-1} + \sum_{i=0}^{n-2} d_i \cdot 2^i & \text{for signed domains} \end{cases} \tag{3.1}$$

Accordingly, the relational operators $=$, $\neq$, $\geq$, and $>$ take their standard meaning in $\mathbb{Z}$. Furthermore, we can define the semantics of addition, subtraction, and multiplication in terms of sequential circuits. The circuit in Figure 3.1, for instance, implements the addition of two bit-vectors of width 3, i.e., $(\mathtt{x} + \mathtt{y} \equiv \mathtt{z}) \mod 8$ where $\mathtt{x}$ is represented by $\mathtt{x}_2\,\mathtt{x}_1\,\mathtt{x}_0$, $\mathtt{y}$ by $\mathtt{y}_2\,\mathtt{y}_1\,\mathtt{y}_0$, and $\mathtt{z}$ by $\mathtt{z}_2\,\mathtt{z}_1\,\mathtt{z}_0$. The unary operator $\sim$ denotes bit-wise negation

$$\sim (d_{n-1} \ldots d_0) \stackrel{\text{def}}{=} (\neg d_{n-1} \ldots \neg d_0)\,,$$

and similarly, the operators `&` and `|` denote bit-wise conjunction and disjunction, respectively. Let c be a numerical constant. The operator $\ll$ denotes a left shift defined as

$$(d_{n-1} \ldots d_0) \ll c \stackrel{\text{def}}{=} (d_{(n-1)-c} \ldots d_0\, 0 \ldots 0) \quad \text{if } c^{\mathcal{M}} < n\,.$$

The operator is treated as an uninterpreted function if $c^{\mathcal{M}} \geq n$. The right shift operator $\gg$ is defined similarly. For details we refer the reader to [KS08] and [BKWW08]. Brinkmann and Drechsler [BD02] provides a formal semantics for the fragment of bit-vector logic presented in Table 3.1.

For reasons of readability, we assume that all values in our domain are represented using bit-vectors of the same width (and all functions use the same modulus, respectively). In our implementation, we lift this restriction by using many-sorted first-order logic. Furthermore, we assume that bit-vectors represent unsigned numbers unless we specify explicitly that they are signed.

Example 3.1.1. *Consider the path*

$$\mathtt{i} := *;\ \mathtt{j} := *;\ [\mathtt{i} > \mathtt{j} + 2];\ \mathtt{assert}(\mathtt{i} > \mathtt{j})$$

and assume that the bit-width of the unsigned variables $\mathtt{i}$ *and* $\mathtt{j}$ *is 32. The strongest postcondition of the prefix reaching the assertion is* $sp(\mathtt{i} := *;\ \mathtt{j} := *;\ [\mathtt{i} > \mathtt{j} + 2], \mathit{true}) = \mathtt{i} > \mathtt{j} + 2$. *Since the predicate* $\mathtt{i} > \mathtt{j}+2 \wedge \mathtt{i} \leq \mathtt{j}$ *has the satisfying assignment* $\{\mathtt{i} \mapsto 2^{32}-1, \mathtt{j} \mapsto 2^{32}-1\}$ *in its bit-vector interpretation, the prefix constitutes a valid counterexample. A decision procedure for the theory of linear arithmetic over the reals or the unbounded integers, however,*

would report that the path is safe.

3.2 Interpolation, Symbol Elimination, and Quantification

This section introduces the concepts the interpolation algorithms in Sections 3.3 to 3.5 are based on. We present a variation of Craig's interpolation theorem adapted to the theory of bit-vector arithmetic. We introduce proofs and restrictions on the structure and nature of these proofs which enable the construction of *partial* interpolants (Section 3.2.3). The interpolation systems in Section 3.3 require the *projection* functions introduced in Section 3.2.3 to generate partial interpolants from propositional resolution proofs, while the algorithms in Section 3.5 require proofs to be *local* (Section 3.2.2). The technique in Section 3.4 relies on partial interpolants to combine the different interpolation systems. The concluding discussion of the relationship between quantification and symbol elimination provides insight into the logical strength of interpolants.

3.2.1 Proofs and Interpolants for Bit-Vector Arithmetic

Following the elaboration of the details of the formal language $\mathcal{L}$ on which we base our verification efforts, we present a variation of Craig's interpolation theorem adapted to our setting. In particular, we extend Theorem 2.2.1 in order to accommodate interpreted and uninterpreted symbols (as introduced in Section 3.1). In accordance with [KV09b] we assume that the meaning of interpreted symbols is determined by an underlying *theory* $\mathcal{T}$. A theory is a set of axioms (closed formulae, respectively), i.e., we assume that $\mathcal{T}$ is axiomatisable in first-order logic. We call the function and predicate symbols occurring in $\mathcal{T}$ interpreted and all other symbols uninterpreted. We use $A_1, \ldots, A_n \models_{\mathcal{T}} A$ to denote that $A_1 \wedge \ldots \wedge A_n \Rightarrow A$ evaluates to true under all assignments satisfying $\mathcal{T}$. The symbol $\vdash_{\mathcal{T}}$ represents a sequence of inference steps.

Definition 3.2.1 (Inference rule)**.** *Given a set of closed formulae $A_1, \ldots, A_n$, and A, an inference rule*

$$\frac{A_1 \quad \cdots \quad A_n}{A},$$

states that the conclusion A can be derived from the conjunction of the premises $A_1 \wedge \ldots \wedge A_n$.

An inference system for $\mathcal{T}$ is a set of inference rules. A derivation in an inference system is a tree built from inferences in the theory $\mathcal{T}$. A proof that a conclusion A follows from a premise in a theory $\mathcal{T}$ is a (finite) derivation in which all leaves are either axioms or elements of the premise and A is the root. A proof showing that false follows from a formula or set of premises is a refutation. Note that $A \models_{\mathcal{T}} B$ does not necessarily imply that $A \vdash_{\mathcal{T}} B$, since the inference system might not be complete for the language $\mathcal{L}$. However, we assume that $A \models_{\mathcal{T}} B$ always follows from $A \vdash_{\mathcal{T}} B$, i.e., that the inference system is sound.

We restate Theorem 2.2.1 such that it accommodates the theory $\mathcal{T}$:

Theorem 3.2.1. *Let A and B two closed formulae in $\mathcal{L}$ (as defined in Section 3.1) and let $\mathcal{T}$ be the respective theory. If $A \models_{\mathcal{T}} B$ holds, then there exists a closed formula I such that $A \models_{\mathcal{T}} I$ and $I \models_{\mathcal{T}} B$ hold, and the uninterpreted function and predicate symbols of I occur in A as well as in B. We call I an interpolant.*

A slightly more general and non-symmetric version of Theorem 3.2.1 and a corresponding proof is presented in [KV09b].

Recall that we use nullary function symbols and predicates to represent program variables in order to retain consistency with the original formulation of Craig's theorem. Note that the language fragment defined by the grammar in Table 3.1 represents the special case in which all function symbols and predicates except the ones used to represent program variables and propositional variables are interpreted. In this setting, the differentiation between unquantified variables and uninterpreted nullary function symbols is solely terminological.

This chapter of our dissertation is concerned with the generation of interpolants in the language of quantifier-free bit-vector arithmetic (as specified in Table 3.1). The existence of efficient satisfiability checking techniques for quantifier-free $\mathcal{L}$-formulae (see [KS08]), which allow us to perform the coverage check for reachability trees (Definition 2.5.1), provide the incentive to restrict ourselves to this fragment. More generally, for many theories (including the theory of arrays) the quantifier-free fragment has a decision procedure to answer the satisfiability queries while the full theory may be undecidable [KMZ06], making quantifier-free formulae particularly interesting.

3.2.2 Local Derivations and Symbol Elimination

Before we proceed to discuss techniques to construct Craig interpolants in Sections 3.3 to 3.5, we point out the close connection between quantification, symbol elimination, and interpolation. Given an interpolant I for a pair of formulae (A, B), it holds by definition (see Theorem 3.2.1) that $A \models_{\mathcal{T}} I$ and $\neg B \models_{\mathcal{T}} \neg I$. Moreover, $A \wedge \neg B$ is unsatisfiable, i.e., $A, \neg B \models_{\mathcal{T}} \mathsf{false}$. This observation gives rise to an alternative formulation of Craig's interpolation theorem (often referred to as Craig-Robinson Theorem [Har09]).

Theorem 3.2.2 (Craig-Robinson Theorem). *Let A and B be two closed formulae in a first-order logic $\mathcal{L}$. If $A \wedge B \models_{\mathcal{T}} \mathsf{false}$, then there exists a closed formula I such that $A \models_{\mathcal{T}} I$ and $B \models_{\mathcal{T}} \neg I$ hold, and the uninterpreted function and predicate symbols of I occur in A as well as in B.*

In [KV09b], the formula I defined in Theorem 3.2.2 is referred to as "reverse" interpolant, since it is a Craig interpolant for $(A, \neg B)$. In this context, we will also refer to $\neg B$ as $\overline{B}$ in order to keep the presentation compact. We claim in Example 2.2.2 that it is possible to derive interpolants for (A, B) from a derivation of false from $(A, \overline{B})$, given that this refutation has a certain structure. The papers [JM06] and [KV09b] state that one can extract an interpolant from a refutation proof if all derivations in the proof are *local*.

Definition 3.2.2 (Local Derivation). *Given a refutation proof for the conjunction of two formulae A and $\overline{B}$, an inference*

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

in this proof is called local *if*

- *either all uninterpreted function and predicate symbols in $C_1, \ldots, C_n$ and C occur in A or all of them occur in $\overline{B}$, and*
- *if all uninterpreted function and predicate symbols in $C_1, \ldots, C_n$ occur in A as well as in $\overline{B}$, then so do the uninterpreted function and predicate symbols of C.*

A derivation, proof, or refutation is called local if each of its inferences is local.

$$\dfrac{\dfrac{\dfrac{\mathtt{y}=\mathtt{x}}{\mathtt{y}\le\mathtt{x}} \qquad \dfrac{\mathtt{y}\ne 0 \qquad \dfrac{\mathtt{y}'=\mathtt{y}\&(\mathtt{y}-\mathtt{1})}{\mathtt{y}'\le \mathtt{y}-\mathtt{1}}}{\mathtt{y}'<\mathtt{y}}}{\mathtt{y}'<\mathtt{x}} \qquad \mathtt{x}=\mathtt{y}'}{\textbf{false}}$$

Figure 3.2: A proof consisting of local derivations

Intuitively, a proof containing only local derivations *eliminates* uninterpreted function and predicate symbols which are local to A or local to $\overline{B}$ before it combines formulae derived from A with formulae derived from $\overline{B}$. Accordingly, no formula in the derivation is allowed to contain a mixture of symbols "local to" A and "local to" $\overline{B}$.

Example 3.2.1. *Consider the formulae* $A \equiv (\mathtt{y} = \mathtt{x} \wedge \mathtt{y} \neq 0 \wedge \mathtt{y}' = \mathtt{y}\&(\mathtt{y}-\mathtt{1}))$ *and* $B \equiv (\mathtt{x} \neq \mathtt{y}')$ *derived from the safe path in Example 2.2.2. Figure 3.2 shows a refutation proof for the conjunction* $A \wedge \overline{B}$*. The sub-tree of the proof encircled with a dashed line contains only function symbols which occur in* A*. The final inference*

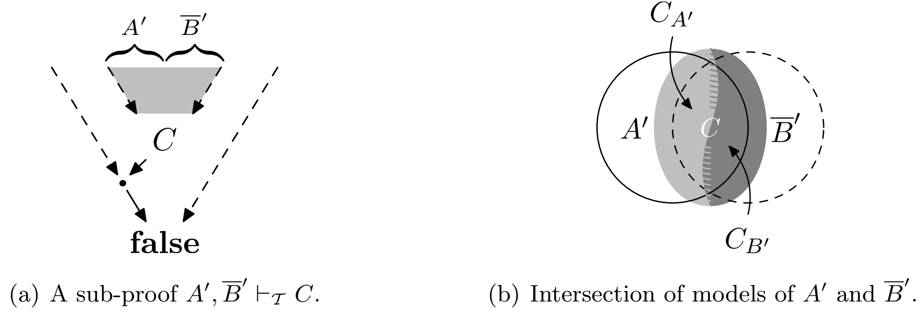
$$\frac{\mathtt{y} \le \mathtt{x} \qquad \mathtt{y}' < \mathtt{y}}{\mathtt{y}' < \mathtt{x}} \ (\textit{transitivity})$$

in the sub-tree eliminates the symbol $\mathtt{y}$ *such that the conclusion refers only to function symbols shared by* A *and* B*.*

3.2.3 Partial Interpolants

Kovács and Voronkov [KV09b] shows that an interpolant for the pair of formulae (A, B) can be extracted in linear time from a refutation proof for $A \wedge \overline{B}$ if every inference in the proof is local. We do not restate their detailed results at this point (we defer this discussion to Section 3.5), but aim at providing an intuition of the concept underlying their construction.

The interpolation system presented in [KV09b] as well as a number of other interpolation procedures (e.g., [Hua95, Kra97, Pud97, McM05, YM05, FGG+09, GKT09]) are based on an inductive definition of interpolants. The algorithms associate a so called *partial* interpolant with each sub-tree of the refutation proof. Consider a sub-tree $A', \overline{B}' \vdash_{\mathcal{T}} C$ of a refutation $A, \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$ (as indicated in Figure 3.3(a)). A' and $\overline{B}'$ represent subsets of the premises



(a) A sub-proof $A', \overline{B}' \vdash_{\mathcal{T}} C$.

(b) Intersection of models of A' and $\overline{B}'$.

Figure 3.3: Illustrations for partial interpolants in refutations

A and $\overline{B}$, respectively.

Since C is a consequence of $A' \wedge \overline{B}'$, C represents a superset of the models that satisfy A' as well as $\overline{B}'$. Therefore, unless $C = \mathsf{false}$, an actual Craig interpolant for A' and B' may not exist because A' may not imply B'. Accordingly, a partial interpolant (or "C-interpolant" in the terminology of [KV09b]) is a slightly "relaxed" notion of a Craig interpolant which takes the conclusion C into account.

It follows immediately from $A', \neg B' \vdash_{\mathcal{T}} C$ that $(A' \wedge \neg C) \Rightarrow (B' \vee C)$. Accordingly, [KV09b] introduces the notion of a "C-interpolant", a formula I which has the following properties:

1. $A' \wedge \neg C \models_{\mathcal{T}} I$,
2. $I \models_{\mathcal{T}} B' \vee C$, and
3. all uninterpreted function and predicate symbols of I occur in A' as well as in B'.

Note that if $A', \overline{B}' \vdash_{\mathcal{T}} C$ is a local derivation, then either all uninterpreted function and predicate symbols in C occur in A or all of them occur in B (cf. Definition 3.2.2). The proof of Lemma 2 in [KV09b] shows that a C-interpolant is a Craig interpolant for the pair of formulae $(A' \wedge \neg C, B')$ in the former case and for $(A', B' \vee C)$ in the latter case. Moreover, if C is false, then the C-interpolant is a Craig interpolant for (A', B') and a reverse interpolant for $(A', \overline{B}')$.

The main result of [KV09b] is that a C-interpolant for a local derivation $A, \overline{B} \vdash_{\mathcal{T}} C$ is a Boolean combination of conclusions and C-interpolants of the sub-proofs of the derivation.

Example 3.2.2. *Consider the local refutation in Figure 3.2. The conclusion* $\mathtt{y}' < \mathtt{x}$ *of the*

local derivation $\mathtt{y} = \mathtt{x}, \mathtt{y} \neq 0, \mathtt{y}' = \mathtt{y}\&(\mathtt{y}-1) \vdash_{\mathcal{T}} \mathtt{y}' < \mathtt{x}$ *is a Craig interpolant for the pair of formulae* $\mathtt{y} = \mathtt{x} \wedge (\mathtt{y} \neq 0) \wedge (\mathtt{y}' = \mathtt{y}\&(\mathtt{y}-1))$ *and* $\mathtt{x} \neq \mathtt{y}'$*.*

In Section 3.5, we present an algorithm which extracts interpolants from local refutations of bit-vector formulae.

The requirement of the locality of derivations is sufficient but not necessary. Yorsh and Musuvathi [YM05] shows that this restriction can be relaxed under certain conditions. [YM05] discusses theories and inference systems which allow splitting the conclusion C of $A', \overline{B}' \vdash_{\mathcal{T}} C$ into an A-part $C_{A'}$ and a B-part $C_{B'}$ (i.e., $\mathrm{Sym}(C_{A'}) \subseteq \mathrm{Sym}(A')$ and $\mathrm{Sym}(C_{B'}) \subseteq \mathrm{Sym}(B')$) such that $C = C_{A'} \vee C_{B'}$. In particular, [YM05] covers the Nelson-Oppen [NO79] framework and the resolution calculus for propositional logic. Note that local derivations also trivially have this property.

Figure 3.3(b) depicts the intersection of the models of two formulae A' and $\overline{B}'$. Intuitively, a fraction of the set of models represented by C can be attributed to $C_{A'}$ (or $C_{B'}$, respectively). Note that these sets of models are not necessarily disjoint (indicated by the shaded area in Figure 3.3(b)). [McM05] and [YM05] introduce a *projection* operation $C|_A$ performing this split for a certain class of formulae C.

Definition 3.2.3 (Projection of Literals)**.** *Let C be a disjunction of literals $\bigvee_{i=1}^{n} l_i$, where each literal l_i is either an atom or a negation of an atom. Given a formula A, $C|_A$ is the disjunction of all literals l_i in C whose uninterpreted function and predicate symbols occur in A (i.e.,* $\mathrm{Sym}(l_i) \subseteq \mathrm{Sym}(A)$*).*

In the case of Nelson-Oppen style derivations $A', \overline{B}' \vdash_{\mathcal{T}} (t_1 = t_2)$ with “AB-pure” conclusions [YM05] (i.e., $\mathrm{Sym}(t_1 = t_2) \subseteq \mathrm{Sym}(A')$ or $\mathrm{Sym}(t_1 = t_2) \subseteq \mathrm{Sym}(B')$ respectively, t_1 and t_2 being terms) the conclusion comprises only a single atom. Given a derivation $A', \overline{B}' \vdash_{\mathcal{T}} C$, we observe that $A' \wedge \neg(C|_{A'}) \Rightarrow B' \vee C|_{B'}$ holds. A partial interpolant is a Craig interpolant for the pair of formulae $A' \wedge \neg(C|_{A'})$ and $B' \vee C|_{B'}$.

Definition 3.2.4 (Partial Interpolant)**.** *Let $A', \overline{B}' \vdash_{\mathcal{T}} C$ be the inference corresponding to a sub-tree in a refutation. Furthermore, let C be a disjunction $\bigvee_{i=1}^{n} l_i$ of literals. A partial interpolant for $A', \overline{B}' \vdash_{\mathcal{T}} C$ is a formula I for which the following conditions hold:*

1. $A' \wedge \neg(C|_{A'}) \models_{\mathcal{T}} I$*,*

2. $I \models_{\mathcal{T}} B' \vee (C|_{B'})$*, and*

3. $\mathrm{Sym}(I) \subseteq \mathrm{Sym}(A') \cap \mathrm{Sym}(B')$*.*

The interpolation techniques discussed in this dissertation map refutation proofs to partial interpolants. In accordance with [DPWK10], we refer to a procedure for constructing an interpolant from a refutation as interpolation system.

Definition 3.2.5 (Interpolation System)**.** *An* interpolation system *is a function that given a refutation* R *of* $A \wedge \overline{B}$ *maps sub-trees of* R *to partial interpolants. An interpolation system is correct if it maps the tree corresponding to* $A, \overline{B} \vdash_{\mathcal{T}}$ **false** *to a Craig interpolant for the pair of formulae* (A, B)*.*

At this point, we intentionally keep the notions of an interpolation system and a partial interpolant general and refine them according to our needs when we discuss interpolation systems for specific inference systems in Sections 3.3 and 3.5. The inference systems discussed in [YM05] impose structural restrictions on the premises in a refutation. Premises in Nelson-Oppen-style derivations are required to be conjunctions of AB-pure literals. Premises of proofs in the propositional resolution calculus are required to be disjunctions of literals.

In Section 3.3 we present an interpolation technique for quantifier-free propositional logic (a fragment of $\mathcal{L}$) which extracts Craig interpolants from propositional resolution refutations using partial interpolants. In Section 3.4 we discuss how partial interpolants can be used to compute Craig interpolants using derivations in different inference systems.

3.2.4 Symbol and Quantifier Elimination

In the following, we investigate the relation between the symbol eliminating inference systems discussed in the previous section and quantifier elimination. This analysis yields a number of general results about the strength and structure of Craig interpolants.

In our setting, all uninterpreted function and predicate symbols of the language of quantifier-free bit-vector arithmetic are used to represent instances of program variables.

Formally, we are not allowed to quantify over these "variables". Nevertheless, we frequently do so in Chapters 1 and 2 in order to construct the strongest postcondition. Moreover, in these previous chapters we take it for granted that these quantified symbols can be eliminated (cf. Section 2.1). In general, however, this is not the case. In the following, we address both issues, starting with the quantification of program variables.

As remarked previously, if we restrict ourselves to the language fragment defined by the grammar in Table 3.1, all uninterpreted function and predicate symbols represent program variables. The use of constant symbols to represent variables is motivated by the fact that Craig's theorem requires the formulae A and B to be closed, i.e., $\mathrm{Var}(A) \cup \mathrm{Var}(B) = \emptyset$. Therefore, for $A, B \in \mathcal{L}$, we may use first-order logic variables to represent program variables as long as all they do not occur unquantified in A or B. All unquantified variables need to be "replaced" by constant symbols, which, in our setting, is a mere matter of terminology – the resulting closed formula and the original formula are equi-satisfiable.

Evidently, in this setting quantification constitutes a means of symbol elimination (at the cost of enriching the language of bit-vector arithmetic with quantifiers). Moreover, this approach allows us to define the *strongest* and *weakest* interpolant (in the implication order) for a pair of formulae (A, B).

Lemma 3.2.1 (Strongest and Weakest Interpolants)**.** *Let (A, B) a pair of closed formulae in the language of bit-vector arithmetic. Furthermore, let σ be a bijective function which maps the nullary function and predicate symbols in A and B to first-order logic variables and propositional variables which do not occur in A and B. We use $\sigma(A)$ ($\sigma(B)$) to denote the substitution of constant and predicate symbols in A (B, respectively) according to σ. Let $A' = \sigma(A)$ and $B' = \sigma(B)$ and let*

$$I'_\exists \equiv \exists \mathrm{x} \in \mathrm{Var}(A') \setminus \mathrm{Var}(B') \, . \, A' \qquad \textit{and} \qquad I'_\forall \equiv \forall \mathrm{x} \in \mathrm{Var}(B') \setminus \mathrm{Var}(A') \, . \, B'$$

denote the formulae in $\mathcal{L}$ in which all variables that occur only in A' (only in B', respectively) are existentially (universally) quantified. Furthermore, let $I_\exists$ denote the formula $I'_\exists$ with all free variables $\mathrm{Var}(I'_\exists)$ replaced by their original counterparts nullary function and predicate symbols again (i.e., $I_\exists = \sigma^{-1}(I'_\exists)$). $I_\forall$ is defined analogously. Then $I_\exists$ is the strongest and

$I_\forall$ is the weakest interpolant in $\mathcal{L}$ for the pair of formulae (A, B).

Proof. We start with justifying the transformation step in which we replace all nullary function and predicate symbols in a closed formula with variables. Let A be a closed formula and let $\mathcal{M}$ be a model of A in $\mathcal{T}$ (denoted as $\mathcal{M} \models_{\mathcal{T}} A$). $\mathcal{M}$ maps every uninterpreted symbol c in A to a value in the domain $\mathcal{C}$ (i.e., $\{c \mapsto c^{\mathcal{M}}\} \in \mathcal{M}$, $c^{\mathcal{M}} \in \mathcal{C}$). Let A' be the formula A with each uninterpreted symbol c replaced by a variable c (where $\sigma(c) = \mathsf{c}$). Moreover, let $\mathcal{M}'$ be a model for A' such that $(\{c \mapsto c^{\mathcal{M}}\} \in \mathcal{M}) \Leftrightarrow (\{\mathsf{c} \mapsto c^{\mathcal{M}}\} \in \mathcal{M}')$ holds, i.e., $\mathcal{M}$ and $\mathcal{M}'$ are isomorphic. Then $\mathcal{M}' \models_{\mathcal{T}} A'$, i.e., for each model satisfying A there exists a corresponding model satisfying A'. A similar claim holds for $I_\exists$ and $I'_\exists$.

Let I be a closed formula constituting an interpolant for the pair of formulae (A, B), i.e., $A \models_{\mathcal{T}} I$ and $I \models_{\mathcal{T}} B$. We show first that $I_\exists \models_{\mathcal{T}} I$. Note that any model of $I'_\exists$ can be extended to a model of A'. Now let $\mathcal{M}$ be a model satisfying $I_\exists$. Due to the isomorphism between models of A and A', $\mathcal{M}$ can be extended to a model $\mathcal{M}^+$ satisfying A. Since $A \models_{\mathcal{T}} I$, it holds that $\mathcal{M}^+ \models_{\mathcal{T}} I$, and since the set of uninterpreted symbols in I is a subset of the function symbols in $I_\exists$, it holds that $\mathcal{M} \vdash_{\mathcal{T}} I$. Accordingly, $I_\exists \models_{\mathcal{T}} I$.

Furthermore, it holds that $I \models_{\mathcal{T}} I_\forall$. To see this, note that $I \models_{\mathcal{T}} B$ can be written as $\neg B \models_{\mathcal{T}} \neg I$. Accordingly, due to an argument similar to the one made above, $\neg I_\forall$ is the strongest formula implied by $\neg B$ in $\mathcal{T}$ such that all uninterpreted symbols of $I_\forall$ also occur in A. Conversely, $I_\forall$ must be the weakest formula implying B, since the set of models satisfying $\neg I_\forall$ is the complement of the set of models of $I_\forall$. □

A variation of Lemma 3.2.1 for propositional logic is provided in [EKS08] (Lemma 1).

Remark 1. The fact that $I \models_{\mathcal{T}} B$ is equivalent to $\neg B \models_{\mathcal{T}} \neg I$, used in the proof of Lemma 3.2.1, immediately leads to the following observation:

Lemma 3.2.2 (Inverse Interpolant)**.** *Let I be a Craig interpolant for the pair of formulae (A, B). Then $\neg I$ is a Craig interpolant for the pair of formulae $(\neg B, \neg A)$.*

Proof. It follows from $A \models_{\mathcal{T}} I$ that $\neg I \models_{\mathcal{T}} \neg A$ and from $I \models_{\mathcal{T}} B$ that $\neg B \models_{\mathcal{T}} \neg I$. □

Similarly, the negation of the reverse interpolant for a pair of formulae $(A, \overline{B})$ is a

reverse interpolant for $(\overline{B}, A)$. Furthermore, observe that the conjunction or disjunction of two Craig interpolants is also an interpolant.

Lemma 3.2.3 (Combination of Interpolants)**.** *Let I_1 and I_2 be Craig interpolants for the pair of formulae (A, B). Then $I_1 \wedge I_2$ as well as $I_1 \vee I_2$ are a Craig interpolants for the pair of formulae (A, B).*

Remark 2. Recall that Lemma 2.2.1 allows us to write $sp(\pi, Q)$ (and $wlp(\pi, Q)$, respectively) as an existentially (universally) quantified formula in prenex normal form (given that Q is a quantifier free predicate). We point out the structural resemblance to the weakest and strongest interpolants (though, as we observe on page 36, not all free variables in $sp(\pi, Q)$ are necessarily free variables of $wlp(\pi, Q)$, and therefore $sp(\pi, Q)$ and $wlp(\pi, Q)$ are not necessarily interpolants for the pair of formulae $(sp(\pi, Q), wlp(\pi, Q))$).

The strongest or weakest Craig interpolant for a pair of formulae (A, B) is not necessarily a formula in the language fragment defined in Table 3.1, since the algorithm outlined in Lemma 2.2.1 may introduce existential and universal quantifiers. Evidently, we are always able to construct quantifier-free interpolants if the underlying theory has quantifier elimination (cf. [KMZ06], Theorem 8).

Example 3.2.3. *Consider the pair of formulae $\mathtt{y} \leq \mathtt{x} \wedge \mathtt{y}' < \mathtt{y}$ and $\mathtt{x} = \mathtt{y}'$ (occurring in the refutation in Figure 3.2). According to Lemma 3.2.1, the strongest Craig interpolant for these formulae is $\exists \mathtt{y} \,.\, \mathtt{y} \leq \mathtt{x} \wedge \mathtt{y}' < \mathtt{y}$ and the weakest interpolant is $\mathtt{x} = \mathtt{y}'$. The transitivity of the inequality relation in the theory of bit-vector arithmetic enables us to obtain a quantifier-free representation $\mathtt{y}' < \mathtt{x}$ of the strongest interpolant.*

A quantified variable $\mathtt{x}$ ranging over a finite domain $\mathcal{C}$ can always be eliminated from a formula $\forall \mathtt{x} \,.\, F$ (or $\exists \mathtt{x} \,.\, F$) by generating a finite conjunction (or disjunction, respectively) of formulae $F[\mathtt{x}/c]$, $c \in \mathcal{C}$. Accordingly, the language fragment in Table 3.1 augmented with quantifiers is amenable to quantifier elimination. In general, however, it is not always possible to eliminate quantifiers.

Table 3.2: The grammar for propositional logic

(a) Propositional Logic

formula	::=	*formula* $\wedge$ *formula* | *formula* $\vee$ *formula* | $\neg$*formula* | (*formula*) | *atom*
atom	::=	*propositional identifier* | *constant*
constant	::=	true | false

(b) Propositional formulae in conjunctive normal form (CNF)

formula	::=	*formula* $\wedge$ (*clause*) | (*clause*)	*clause*	::=	*clause* $\vee$ *literal* | *literal*
literal	::=	*atom* | $\neg$*atom*	*atom*	::=	*propositional identifier*

Example 3.2.4. *The strongest interpolant for the pair of formulae* $\mathtt{x} = 2\cdot\mathtt{z}$ *and* $\mathtt{x} \neq 2\cdot\mathtt{y}+1$ *(where* x, y, z *take values in the unbounded domain* $\mathbb{Z}$*) is* $\exists \mathtt{z}\,.\,\mathtt{x} = 2\cdot\mathtt{z}$*. There is no quantifier-free interpolant expressing that* x *is even in the language of linear arithmetic over the integers [McM05, JCG09].*

Finally, given a local refutation $A \wedge \neg B \vdash_{\mathcal{T}}$ false, it is not necessarily possible to derive the strongest and weakest interpolant for the pair of formulae (A, B). Intuitively, the structure of the proof dictates a nesting of quantifiers. In particular, two different instances of the same symbol may be eliminated in different sub-trees of the proof. The nesting determines the strength of the interpolant. In certain cases, this restriction can be lifted by modifying the proof. Such proof transformations are discussed in Section 3.3.5.

3.3 Interpolation and Propositional Logic

Efficient solvers for bit-vector arithmetic often reduce the problem instances to a language fragment of $\mathcal{L}$ for which efficient proof-generating decision procedures are available. Propositional logic (see Table 3.2(a)) is a prominent example of such a target language (see, for instance, [DdM06, BKO+07, JLS09]). A number of proof-generating satisfiablity solvers for a propositional formulae in conjunctive normal form (CNF) is available (e.g., [ZM03, ES04, Bie08]). CNF is the fragment of propositional logic specified by the grammar in Table 3.2(b) to which every propositional formula can be reduced by means of a satisfiability-preserving transformation.

3.3.1 Flattening Bit-Vector Formulae

Due to the finitary nature of bit-vector arithmetic, every bit-vector formula can be represented by means of an equi-satisfiable formula in propositional logic. In this section, we describe a transformation commonly known as flattening or bit-blasting.

The propositional fragment in Table 3.2(a) lacks the theory-specific atoms of bit-vector arithmetic (i.e., the non-terminal *term* $\triangleright$ *term* in Table 3.1 is not part of the propositional language). Given a bit-vector formula F, we use $\text{Atoms}(F)$ to denote the atoms occurring in F, and $\text{Atoms}^{\mathcal{T}}(F)$ to denote its theory-specific atoms (i.e., $\text{Atoms}^{\mathcal{T}}(F) \subseteq \text{Atoms}(F)$). Propositional logic is sufficiently expressive to represent the propositional structure of a bit-vector arithmetic formula.

Definition 3.3.1 (Propositional Skeleton)**.** *Let F be a formula in the bit-vector logic language presented in Table 3.1. The propositional skeleton of F is denoted by $\mathsf{sk}(F)$ and is obtained by replacing every atom that is not a propositional identifier by a fresh propositional identifier.*

Observe that $\mathsf{sk}(F)$ determines a bijective mapping between the atoms in $\text{Atoms}(\mathsf{sk}(F))$ and the atoms $\text{Atoms}(F)$. In this chapter, we use e_i to denote a propositional atom and α_i to denote the corresponding theory atom. (Multiple occurrences of α_i are always replaced by the same atom e_i.) The propositional skeleton $\mathsf{sk}(F)$ is an approximation of F inasmuch as it preserves satisfiablity. If F is satisfiable, then so is $\mathsf{sk}(F)$, whereas the converse does not necessarily hold.

Example 3.3.1. *The propositional skeleton of the bit-vector formula*

$$(\neg(\mathsf{x} = \mathsf{y}) \vee ((\mathsf{x}\&2) = 2)) \wedge (\mathsf{y} = \mathsf{z} + \mathsf{z}) \wedge (\mathsf{x} = \mathsf{z} \ll 1) \wedge ((\mathsf{z}\&1) = 0) \tag{3.2}$$

is

$$(\neg \mathsf{e}_1 \vee \mathsf{e}_2) \wedge \mathsf{e}_3 \wedge \mathsf{e}_4 \wedge \mathsf{e}_5\,. \tag{3.3}$$

*The assignment which maps all propositional atoms in $\mathsf{sk}(F)$ to **true** (represented by $\bigwedge_{i=1}^{5} \mathsf{e}_i$) satisfies the Formula (3.3), whereas the Formula (3.2) is unsatisfiable.*

The accuracy of the approximation $\mathsf{sk}(F)$ can be further improved by adding contraints which reflect the semantics of the theory atoms $\text{Atoms}^{\mathcal{T}}(F)$ for each respective propositional atom in $\mathsf{sk}(F)$. In the following, we sketch the propositional encoding of the bit-vector operators in Table 3.1. A more detailed description of the encoding is provided in [BKWW08] and [KS08].

We perform an initial transformation which guarantees that the parameters of a bit-vector operation are always symbols. Given a term $t_1 \bigcirc t_2$ in a formula, we replace the sub-terms t_1 and t_2 with fresh symbols $\mathtt{z}_1$ and $\mathtt{z}_2$, respectively, and add the constraint $(\mathtt{z}_1 = t_1) \wedge (\mathtt{z}_2 = t_2)$ to the resulting new formula.

Let n denote the bit-width of the program variables $\mathtt{x}$ and $\mathtt{y}$ in a term $\mathtt{x} \bigcirc \mathtt{y}$, where $\bigcirc$ represents an operation in $\{+, -, \cdot, \&, \mid, \oplus, \ll, \gg\}$. Then, the symbols $\mathtt{x}$ and $\mathtt{y}$ are represented by n propositional symbols $\mathtt{x}_{n-1}, \ldots, \mathtt{x}_0$ and $\mathtt{y}_{n-1}, \ldots, \mathtt{y}_0$, respectively. As explained above, we introduce a fresh symbol $\mathtt{z}$ and n corresponding propositional symbols $\mathtt{z}_{n-1}, \ldots, \mathtt{z}_0$ representing the term $\mathtt{x} \bigcirc \mathtt{y}$.

Bit-wise operations. The bit-wise operations $\mathtt{z} = \mathtt{x} \,\&\, \mathtt{y}$, $\mathtt{z} = \mathtt{x} \mid \mathtt{y}$, and $\mathtt{z} = \mathtt{x} \oplus \mathtt{y}$ are encoded as

$$\bigwedge_{i=0}^{n-1} (\mathtt{z}_i \Leftrightarrow (\mathtt{x}_i \wedge \mathtt{y}_i)), \quad \bigwedge_{i=0}^{n-1} (\mathtt{z}_i \Leftrightarrow (\mathtt{x}_i \vee \mathtt{y}_i)), \quad \text{and} \quad \bigwedge_{i=0}^{n-1} \mathtt{z}_i \Leftrightarrow ((\mathtt{x}_i \vee \mathtt{y}_i) \wedge (\neg \mathtt{x}_i \vee \neg \mathtt{y}_i)),$$

respectively, and the bit-wise negation $\mathtt{z} = \sim \mathtt{x}$ is represented by $\bigwedge_{i=0}^{n-1} (\mathtt{z}_i \Leftrightarrow \neg \mathtt{x}_i)$.

Shift operations. Shift-operations are implemented by means of a cascade of parallel multiplexers known as barrel shifter. Figure 3.4 shows the circuit diagram of a 4-bit barrel shifter implementing the operation $\mathtt{z} = \mathtt{x} \ll \mathtt{y}$. The ith stage of the shifter performs a shift by 2^i positions if $\mathtt{y}_i$ is true.

Arithmetic operations. Figure 3.1 shows the implementation of a carry-look-ahead adder for two inputs $\mathtt{x}$ and $\mathtt{y}$ of bit-width three. Subtraction is enabled by the bit-wise negation of the subtrahend $\mathtt{y}$ and setting the carry-in flag $\mathtt{c}_{in}$ to true, which corresponds to taking the two's complement of $\mathtt{y}$. Multiplication can be encoded using a shift-and-add

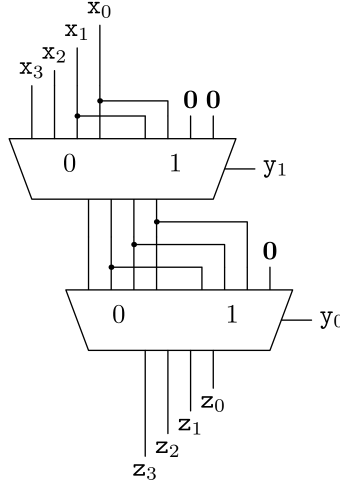


Figure 3.4: The implementation of a 4-bit barrel shifter

circuit, i.e., the multiplication of two 2-bit parameters $\mathtt{x}$ and $\mathtt{y}$ represented as $[\mathtt{x}_1\,\mathtt{x}_0]$ and $[\mathtt{y}_1\,\mathtt{y}_0]$, respectively, is defined as

$$[\mathtt{z}_2\,\mathtt{z}_1\,\mathtt{z}_0] = ([0\,\mathtt{x}_1\,\mathtt{x}_0]\&[\mathtt{y}_0\,\mathtt{y}_0\,\mathtt{y}_0]) + (([0\,\mathtt{x}_1\,\mathtt{x}_0] \ll 1)\&[\mathtt{y}_1\,\mathtt{y}_1\,\mathtt{y}_1]) \ .$$

Notably, it is possible to represent integer division $\mathtt{z} = \frac{\mathtt{x}}{\mathtt{y}}$ by means of its inverse operation multiplication using the constraint $(\mathtt{z} \cdot \mathtt{y} + \mathtt{r} = \mathtt{x}) \wedge (\mathtt{r} < \mathtt{y})$ (for $\mathtt{y} \neq 0$), where $\mathtt{r}$ denotes the remainder.

Relational Operators. Having defined the encoding of the terms in the grammar in Table 3.1, we proceed to the encoding of theory-specific atoms *term* $\rhd$ *term*, where $\rhd$ is either $=$ or $>$ (the relations $\geq$ and $\neq$ can be encoded in terms of $>$, $=$ and Boolean operations). The encoding of the equality $\mathtt{x} = \mathtt{y}$ is straight-forward:

$$\bigwedge_{i=0}^{n-1} (\mathtt{x}_i \Leftrightarrow \mathtt{y}_i)$$

The inequality $\mathtt{x} < \mathtt{y}$ can be expressed in terms of subtraction. If $\mathtt{x} < \mathtt{y}$ holds, then the subtraction $\mathtt{x} - \mathtt{y}$ yields an overflow, which can be detected by checking the signals c_{out}, $\mathtt{x}_2$, and $\mathtt{y}_2$ in Figure 3.1. For unsigned operands, an overflow occurs if $\mathsf{c}_{out} = \textsf{true}$. In case of signed operands, the value of $(\mathtt{x}_2 \Leftrightarrow \mathtt{y}_2) \Leftrightarrow \mathsf{c}_{out}$ indicates whether an overflow occurred.

We use $\mathcal{E}(\alpha_i)$ to denote the propositional encoding of a theory atom α_i. Given a propositional symbol $\mathsf{e} \in \text{Atoms}(\mathsf{sk}(F))$, contemporary decision procedures construct a constraint

$$(\mathsf{e} \Rightarrow \mathcal{E}(\alpha)) \wedge (\neg\mathsf{e} \Rightarrow \neg\mathcal{E}(\alpha)) \tag{3.4}$$

encoding the semantics of the theory atom α. The solvers expose various levels of aggressiveness when it comes to constraining the atoms of the propositional skeleton. *Lazy* approaches iteratively strengthen the encoding, much like the iterative refinement approach presented in Section 2.5.4. *Eager* flattening techniques perform the transformation in a single step. In the context of this dissertation, we are solely interested in the resulting propositional representation of bit-vector formulae and refer the reader to [KS08] for a more detailed discussion of eager and lazy encoding strategies.

Conjunctive Normal Form. Most satisfiability solvers for propositional formulae expect their input to be presented in CNF (as specified in Table 3.2(b)). A formula in CNF comprises literals (which are propositional atoms or their negations) and clauses (which are disjunctions of literals). A formula in CNF is a conjunction of clauses, often also represented as set of clauses. We write $l \in C$ if a literal l occurs in a clause C, and $C \in A$ if a clause C occurs in a formula A which is in CNF.

The encoding obtained by means of flattening as described previously does not adhere to the syntactical restrictions imposed the grammar in Table 3.2(b). It is, however, possible to transform the flattened formula into an equi-satisfiable formula in CNF by means of Tseitin's encoding [Tse83]. Analogously to the pre-processing step which replaces sub-terms t_i with fresh symbols $\mathtt{z}_i$, Tseitin's encoding introduces a fresh propositional symbol and a corresponding constraint for each sub-formula. This transformation results in a conjunction of sub-formulae of the form $\mathtt{x} \Leftrightarrow R(\mathtt{y}, \mathtt{z})$, where R represents a Boolean combination of $\mathtt{y}$ and $\mathtt{z}$. Each of these sub-formulae is then converted into an equivalent formula in CNF. Table 3.3 shows the respective transformation rules for the propositional structures introduced by flattening the bit-vector formula. Tseitin's transformation results in a linear increase of the size of the formula (at the cost of introducing new propositional symbols).

Negation:

$$\begin{aligned}\mathtt{x} \Leftrightarrow \neg\mathtt{y} &\equiv (\mathtt{x} \Rightarrow \neg\mathtt{y}) \wedge (\neg\mathtt{y} \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \neg\mathtt{y}) \wedge (\mathtt{y} \vee \mathtt{x})\end{aligned}$$

Disjunction:

$$\begin{aligned}\mathtt{x} \Leftrightarrow (\mathtt{y} \vee \mathtt{z}) &\equiv (\mathtt{y} \Rightarrow \mathtt{x}) \wedge (\mathtt{z} \Rightarrow \mathtt{x}) \wedge (\mathtt{x} \Rightarrow (\mathtt{y} \vee \mathtt{z}))\\ &\equiv (\neg\mathtt{y} \vee \mathtt{x}) \wedge (\neg\mathtt{z} \vee \mathtt{x}) \wedge (\neg\mathtt{x} \vee \mathtt{y} \vee \mathtt{z})\end{aligned}$$

Conjunction:

$$\begin{aligned}\mathtt{x} \Leftrightarrow (\mathtt{y} \wedge \mathtt{z}) &\equiv (\mathtt{x} \Rightarrow \mathtt{y}) \wedge (\mathtt{x} \Rightarrow \mathtt{z}) \wedge ((\mathtt{y} \wedge \mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \mathtt{y}) \wedge (\neg\mathtt{x} \vee \mathtt{z}) \wedge (\neg(\mathtt{y} \wedge \mathtt{z}) \vee \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \mathtt{y}) \wedge (\neg\mathtt{x} \vee \mathtt{z}) \wedge (\neg\mathtt{y} \vee \neg\mathtt{z} \vee \mathtt{x})\end{aligned}$$

Equivalence:

$$\begin{aligned}\mathtt{x} \Leftrightarrow (\mathtt{y} \Leftrightarrow \mathtt{z}) &\equiv (\mathtt{x} \Rightarrow (\mathtt{y} \Leftrightarrow \mathtt{z})) \wedge ((\mathtt{y} \Leftrightarrow \mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\mathtt{x} \Rightarrow ((\mathtt{y} \Rightarrow \mathtt{z}) \wedge (\mathtt{z} \Rightarrow \mathtt{y})) \wedge ((\mathtt{y} \Leftrightarrow \mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\mathtt{x} \Rightarrow (\mathtt{y} \Rightarrow \mathtt{z})) \wedge (\mathtt{x} \Rightarrow (\mathtt{z} \Rightarrow \mathtt{y})) \wedge ((\mathtt{y} \Leftrightarrow \mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \neg\mathtt{y} \vee \mathtt{z}) \wedge (\neg\mathtt{x} \vee \neg\mathtt{z} \vee \mathtt{y}) \wedge ((\mathtt{y} \Leftrightarrow \mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \neg\mathtt{y} \vee \mathtt{z}) \wedge (\neg\mathtt{x} \vee \neg\mathtt{z} \vee \mathtt{y}) \wedge (((\mathtt{y} \wedge \mathtt{z}) \vee (\neg\mathtt{y} \wedge \neg\mathtt{z})) \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \neg\mathtt{y} \vee \mathtt{z}) \wedge (\neg\mathtt{x} \vee \neg\mathtt{z} \vee \mathtt{y}) \wedge ((\mathtt{y} \wedge \mathtt{z}) \Rightarrow \mathtt{x}) \wedge ((\neg\mathtt{y} \wedge \neg\mathtt{z}) \Rightarrow \mathtt{x})\\ &\equiv (\neg\mathtt{x} \vee \neg\mathtt{y} \vee \mathtt{z}) \wedge (\neg\mathtt{x} \vee \neg\mathtt{z} \vee \mathtt{y}) \wedge (\neg\mathtt{y} \vee \neg\mathtt{z} \vee \mathtt{x}) \wedge (\mathtt{y} \vee \mathtt{z} \vee \mathtt{x})\end{aligned}$$

Table 3.3: Tseitin transformation [Tse83] for standard Boolean connectives [BKWW08]

Example 3.3.2. *Consider the propositional encoding* $\mathtt{e}_1 \Leftrightarrow \bigwedge_{i=0}^{n-1}(\mathtt{x}_i \Leftrightarrow \mathtt{y}_i)$ *of the constraint* $\mathtt{e}_1 \Leftrightarrow (\mathtt{x} = \mathtt{y})$ *for the skeleton in Example 3.3.1. The bi-implication can be rewritten as*

$$\bigwedge_{i=0}^{n-1}(\neg\mathtt{e}_1 \vee (\mathtt{x}_i \Leftrightarrow \mathtt{y}_i)) \quad \wedge \quad \left(\mathtt{e}_1 \vee \bigvee_{i=0}^{n-1}((\mathtt{x}_i \wedge \neg\mathtt{y}_i) \vee (\neg\mathtt{x}_i \wedge \mathtt{y}_i))\right) \tag{3.5}$$

The left side of this formula can be rewritten in CNF as

$$\bigwedge_{i=0}^{n-1}((\neg\mathtt{e}_1 \vee \neg\mathtt{x}_i \vee \mathtt{y}_i) \wedge (\neg\mathtt{e}_1 \vee \mathtt{x}_i \vee \neg\mathtt{y}_i)) \ .$$

By introducing a fresh symbol $\mathtt{o}_i$ *for each of the conjuncts* $(\mathtt{x}_i \wedge \neg\mathtt{y}_i)$ *and* $(\neg\mathtt{x}_i \wedge \mathtt{y}_i)$*, respectively, we obtain*

$$(\mathtt{e_i} \vee \mathtt{o}_0 \vee \ldots \vee \mathtt{o}_{2\cdot n-1}) \wedge \bigwedge_{i=0}^{n-1}((\mathtt{o}_i \Leftrightarrow (\mathtt{x}_i \wedge \neg\mathtt{y}_i)) \ \wedge \ (\mathtt{o}_{i+n} \Leftrightarrow (\neg\mathtt{x}_i \wedge \mathtt{y}_i))) \ ,$$

Encoding	Propositional constraint	CNF clauses
$\mathtt{e}_1 \Leftrightarrow (\mathtt{x} = \mathtt{y})$	$\mathtt{e}_1 \Rightarrow (\mathtt{x}_0 \Leftrightarrow \mathtt{y}_0) \wedge \mathtt{e}_1 \Rightarrow \ldots \wedge \left(\bigwedge_{i=0}^{n-1} \mathtt{x}_i \Leftrightarrow \mathtt{y}_i\right) \Rightarrow \mathtt{e}_1$	$(\neg\mathtt{e}_1 \vee \neg\mathtt{x}_0 \vee \mathtt{y}_0) \wedge (\neg\mathtt{e}_1 \vee \mathtt{x}_0 \vee \neg\mathtt{y}_0) \wedge \ldots \wedge (\mathtt{e_i} \vee \mathtt{o}_0 \vee \ldots \vee \mathtt{o}_{2\cdot n-1}) \wedge \ldots$
$\mathtt{e}_2 \Leftrightarrow ((\mathtt{x}\&2) = 2)$	$\mathtt{e}_2 \Rightarrow \mathtt{x}_1$	$(\neg\mathtt{e}_2 \vee \mathtt{x}_1)$
$\mathtt{e}_3 \Leftrightarrow (\mathtt{y} = \mathtt{z} + \mathtt{z})$	$\mathtt{e}_3 \Rightarrow \neg\mathtt{y}_0 \wedge \ldots$	$(\neg\mathtt{e}_3 \vee \neg\mathtt{y}_0) \wedge \ldots$
$\mathtt{e}_4 \Leftrightarrow (\mathtt{x} = \mathtt{z} \ll 1)$	$(\mathtt{e}_4 \Rightarrow (\mathtt{x}_1 \Leftrightarrow \mathtt{z}_0)) \wedge (\mathtt{e}_4 \Rightarrow \neg\mathtt{x}_0) \wedge \ldots$	$(\neg\mathtt{e}_4 \vee \neg\mathtt{x}_1 \vee \mathtt{z}_0) \wedge (\neg\mathtt{e}_4 \vee \mathtt{x}_1 \vee \neg\mathtt{z}_0) \wedge (\neg\mathtt{e}_4 \vee \neg\mathtt{x}_0) \wedge \ldots$
$\mathtt{e}_5 \Leftrightarrow ((\mathtt{z}\&1) = 0)$	$\mathtt{e}_5 \Rightarrow \neg\mathtt{z}_0$	$(\neg\mathtt{e}_5 \vee \neg\mathtt{z}_0)$

Table 3.4: A propositional encoding of the constraints for the skeleton in Example 3.3.1

which replaces the right part of Formula (3.5). The conjuncts $(\mathtt{o}_i \Leftrightarrow (\mathtt{x}_i \wedge \neg\mathtt{y}_i))$ *and* $(\mathtt{o}_{i+n} \Leftrightarrow (\neg\mathtt{x}_i \wedge \mathtt{y}_i))$ *can be replaced by formulae in CNF according to Table 3.3.*

With a CNF encoding of the constraints derived from the theory atoms in place, we can proceed to refine the propositional skeleton.

Example 3.3.3. *We continue working in the setting of Example 3.3.1 and add a set of propositional constraints which rules out the satisfiable assignment* $\bigwedge_{i=1}^{5} \mathtt{e}_i$*. First, consider the theory atoms* $\mathtt{x}\&2 = 2$ *and* $\mathtt{z}\&1 = 0$*, which dictate that* $\mathtt{x}_1 =$ ***true*** *and* $\mathtt{z}_0 =$ ***false****. This is encoded in the propositional constraints for* $\mathtt{e}_2$ *and* $\mathtt{e}_5$ *in Table 3.4. Moreover, the shift operation in the theory atom* $\mathtt{x} = \mathtt{z} \ll 1$ *forces* $\mathtt{x}_1$ *and* $\mathtt{z}_0$ *to be equal and* $\mathtt{x}_0$ *to be* ***false****. This contradicts* $(\mathtt{x}\&2) = 2 \wedge (\mathtt{z}\&1) = 0$*. Therefore, the constraints eliminate any assignment consistent with* $\mathtt{e}_2 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5$*.*

The propositional skeleton presented in Example 3.3.1 is already in CNF and requires no further transformation.

3.3.2 Resolution Refutations

Resolution is a rule of inference (see Definition 3.2.1) enabling the construction of refutation proofs for unsatisfiable formulae in CNF. The resolution rule eliminates symbols and enables us to construct proofs adhering to the structural restrictions discussed in Section 3.2.1. Let C and D represent propositional clauses (as defined in Table 3.2(b)) and let $\mathtt{x}$ be a

propositional atom. The resolution rule is formally defined below.

$$\frac{(C \vee \mathtt{x}) \qquad (D \vee \neg\mathtt{x})}{(C \vee D)} \;\textsf{Res} \tag{3.6}$$

We refer to $\mathtt{x}$ as the pivot and to $(C \vee D)$ as the resolvent of the resolution. Furthermore, we use $\mathrm{Res}((C\vee\mathtt{x}), (D\vee\neg\mathtt{x}), \mathtt{x})$ to refer to the resolvent of $(C\vee\mathtt{x})$ and $(D\vee\neg\mathtt{x})$ with respect to the pivot $\mathtt{x}$. The rules of associativity and commutativity are applied implicitly. Then, the inference system comprising only the resolution rule is refutation complete for formulae in CNF, i.e., if $A \models_{\mathcal{T}} \textsf{false}$ holds for a formula A in CNF, then so does $A \vdash_{\mathcal{T}} \textsf{false}$.

A refutation $A \vdash_{\mathcal{T}} \textsf{false}$ in the resolution-based inference system is a tree each leaf node of which corresponds to a clause of A and each inner node of which corresponds to a resolution step. A more compact representation of resolution refutations are directed acyclic graphs, which enable sharing of clauses between resolution steps.

Definition 3.3.2. *A resolution proof R is a directed acyclic graph $(V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$, where V_R is a set of vertices, E_R is a set of edges, piv_R and ℓ_R are functions assigning pivots and clauses to vertices, respectively, and $\mathrm{s}_R \in V_R$ is a unique vertex (called the sink of R) with out-degree zero. An initial vertex has in-degree 0. All other vertices are internal and have in-degree 2. For each internal vertex $v \in V_R$, there are predecessor nodes v^+ and v^- (i.e., $(v^+, v) \in E_R$ and $(v^-, v) \in E_R$) such that $\ell_R(v^+) = (C \vee piv(v))$ and $\ell_R(v^-) = (D \vee \neg piv(v))$, and it holds that*

$$\frac{\ell_R(v^+) \qquad \ell_R(v^-)}{\ell_R(v)} \;\textsf{Res}\,.$$

A resolution proof R is a refutation if $\ell_R(\mathrm{s}_R) = \square$ (where $\square$ represents the empty clause which is logically equivalent to **false***). A refutation R is a refutation of a CNF formula A if for each initial vertex $v \in V_R$ it holds that $\ell_R(v)$ is a clause in A.*

We drop the subscripts in Definition 3.3.2 if it is obvious from the context to which proof R we refer. In order to allow for a compact representation of resolution proofs, we use $\overline{\mathtt{x}}$ as an alternative notation for $\neg\mathtt{x}$, $\top$ and $\bot$ for $\textsf{true}$ and $\textsf{false}$, respectively, and $l_1\, l_2\, \ldots\, l_n$

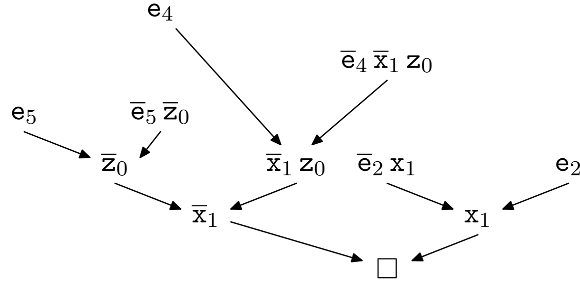


Figure 3.5: A resolution proof for the formula presented in Example 3.3.4

to represent a clause $(l_1 \vee l_2 \vee \ldots \vee l_n)$ over the literals $l_1, \ldots, l_n$.

Example 3.3.4. *Figure 3.5 shows a resolution proof for the formula*

$$e_2 \wedge e_4 \wedge e_5 \;\wedge\; (\neg e_2 \vee x_1) \;\wedge\; (\neg e_4 \vee \neg x_1 \vee z_0) \;\wedge\; (\neg e_5 \vee \neg z_0)\,.$$

An informal justification for the unsatisfiability of this formula is provided in in Example 3.3.3. Note that in Figure 3.5 we use the more compact representation of clauses, i.e., we write $(\overline{e}_4\,\overline{x}_1\,z_0)$ *instead of* $(\neg e_4 \vee \neg x_1 \vee z_0)$.

3.3.3 Interpolation Systems for Propositional Resolution Proofs

In this section, we discuss interpolation systems that enable us to construct a Craig interpolant for (A, B) from a resolution refutation of $A \wedge \overline{B}$. This section covers the approach of Huang, Krajíček, and Pudlák [Hua95, Kra97, Pud97], who were the first to present a linear-time interpolation system based on propositional resolution proofs. Furthermore, we discuss a variation of their algorithm proposed by McMillan in [McM05]. Finally, we present a technique that generalises the interpolation systems mentioned above and enables us to generate interpolants of different logical strength. This technique has been previously published by the authors and his collaborators in [DPWK10].[1]

An interpolation system is a procedure for constructing a Craig interpolant for the pair of formulae (A, B) from a refutation $A, \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$. This section is exclusively concerned

[1]The technique is based on Mitra Purandare's observation that predicate symbols occurring in A which are eliminated in derivations not involving premises from B need not occur in the interpolant for (A, B). The formal interpolation system presented in this dissertation was devised by the author in close cooperation with his collaborator Vijay D'Silva.

with resolution refutations. All interpolation systems discussed in this section are based on an inductive definition of partial interpolants (a concept introduced in Section 3.2.3). The slightly altered setting necessitates an adaptation of the partial interpolants introduced in Definition 3.2.4, which we defer until after the definition of our interpolation systems for resolution proofs (see page 108). Similarly, we refine the notion of an interpolation system (Definition 3.2.5) according to our setting.

Definition 3.3.3 (Interpolation System for Propositional Resolution Refutation)**.** *An interpolation system* Itp *for propositional resolution refutations is a function that given a refutation* $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ *of a formula* $A \wedge \overline{B}$ *in CNF maps each vertex* $v \in V_R$ *to a partial interpolant for* v*. Given a refutation* R *representing* $A, \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$ *we use* $\mathsf{Itp}(R, A, \overline{B})$ *to denote the mapping from* V_R *to partial interpolants.*

An interpolation system Itp *for propositional refutations is correct if for every refutation* R *of* $A \wedge \overline{B}$ *it holds that* $\mathsf{Itp}(R, A, \overline{B})(\mathrm{s}_R)$ *(where* $\ell(\mathrm{s}_R) = \square$*) is a Craig interpolant for* (A, B)*.*

We write $\mathsf{Itp}(R)$ for $\mathsf{Itp}(R, A, \overline{B})$ when the formulae A, $\overline{B}$ are clear from the context. An interpolation system Itp generates an *annotated* resolution refutation. The mapping $\mathsf{Itp}(R, A, \overline{B})$ associates a partial interpolant $\mathsf{Itp}(R, A, \overline{B})(v)$ with each vertex $v \in V_R$ and its respective clause clause $\ell(v)$. In accordance with [McM05], we use the notation $\ell(v) \quad [\,\mathsf{Itp}(R, A, \overline{B})(v)\,]$ to represent this annotation.

Note that, given a refutation R of a formula $A \wedge \overline{B}$, $\mathsf{Itp}(R, A, \overline{B})(\mathrm{s}_R)$ is a reverse interpolant for $(A, \overline{B})$ (i.e., an interpolant in the sense of Theorem 3.2.2) and a Craig interpolant for (A, B).

In the following, we discuss specific instances of interpolation systems for propositional resolution refutations, starting with the system introduced by Huang, Krajíček, and Pudlák.

Definition 3.3.4 (Interpolation System by Huang, Krajíček, and Pudlák)**.** *The interpolation system* Itp_{HKP} *maps each vertex of a resolution refutation* $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ *of* $A \wedge \overline{B}$ *to a partial interpolant according to the following rules:*

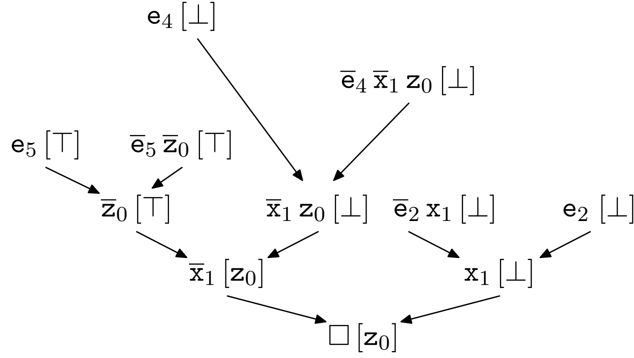


Figure 3.6: Annotations for the resolution proof in Figure 3.5 generated using $\mathsf{Itp}_{\mathrm{HKP}}$

1. For each initial vertex v let

$$\mathsf{Itp}_{HKP}(R, A, \overline{B})(v) \stackrel{\text{def}}{=} \begin{cases} \textsf{\textit{false}} & \text{if } \ell(v) \in A \\ \textsf{\textit{true}} & \text{if } \ell(v) \in \overline{B} \end{cases} .$$

2. Otherwise, v is an internal vertex with $(v^+, v) \in E_R$ and $(v^-, v) \in E_R$. Thus, $\ell(v)$ is the conclusion of a resolution step with the premises $\ell(v^+) = C \vee \mathtt{x}$ and $\ell(v^-) = D \vee \neg\mathtt{x}$ and the pivot $piv_R(v) = \mathtt{x}$. Then

$$\mathsf{Itp}_{HKP}(R, A, \overline{B})(v) \stackrel{\text{def}}{=} \begin{cases} I_1 \vee I_2 & \text{if } \mathtt{x} \in \mathrm{Atoms}(A) \setminus \mathrm{Atoms}(B) \\ (\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2) & \text{if } \mathtt{x} \in \mathrm{Atoms}(A) \cap \mathrm{Atoms}(B) \\ I_1 \wedge I_2 & \text{if } \mathtt{x} \in \mathrm{Atoms}(B) \setminus \mathrm{Atoms}(A) \end{cases} ,$$

where

$$I_1 = \mathsf{Itp}_{HKP}(R, A, \overline{B})(v^+) \quad \text{and} \quad I_2 = \mathsf{Itp}_{HKP}(R, A, \overline{B})(v^-) .$$

A proof of the correctness of this interpolation system is provided in [YM05] and establishes that each annotation generated by $\mathsf{Itp}_{\mathrm{HKP}}$ is indeed a partial interpolant according to Definition 3.2.4.

Example 3.3.5. *We continue working in the setting of Example 3.3.4. We split the formula presented there into the two partitions*

$$A \equiv \mathsf{e}_2 \wedge \mathsf{e}_4 \ \wedge \ (\neg\mathsf{e}_2 \vee \mathtt{x}_1) \ \wedge \ (\neg\mathsf{e}_4 \vee \neg\mathtt{x}_1 \vee \mathsf{z}_0) \quad \text{and} \quad \overline{B} \equiv \mathsf{e}_5 \ \wedge \ (\neg\mathsf{e}_5 \vee \neg\mathsf{z}_0)$$

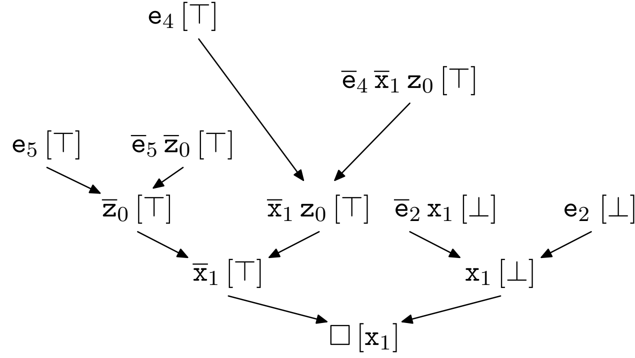


Figure 3.7: Annotations for the resolution proof in Figure 3.5 generated using $\mathsf{Itp}_{\mathrm{HKP}}$ for the partitioning in Example 3.3.6

and annotate the resolution refutation in Figure 3.5 with the partial interpolants generated by the interpolation system $\mathsf{Itp}_{HKP}(R, A, \overline{B})$ *(see Figure 3.6). The resulting reverse interpolant is* $\mathsf{Itp}_{HKP}(R, A, \overline{B})(\mathrm{s}_R) = \mathsf{z}_0$. *It can easily be verified that* z_0 *is a Craig interpolant for*

$$\mathsf{e}_2 \wedge \mathsf{e}_4 \;\wedge\; (\neg\mathsf{e}_2 \vee \mathtt{x}_1) \;\wedge\; (\neg\mathsf{e}_4 \vee \neg\mathtt{x}_1 \vee \mathsf{z}_0) \quad \textit{and} \quad \neg\mathsf{e}_5 \;\vee\; (\mathsf{e}_5 \wedge \mathsf{z}_0)\,.$$

The propositional interpolant z_0 *corresponds to the negation of the constraint contributed by the theory atom* $\mathtt{z}\&1 = 0$ *in Example 3.3.3.*

Observe that the parameters A and $\overline{B}$ of $\mathsf{Itp}(R, A, \overline{B})$ merely define a partitioning of the premises. A refutation $A \wedge \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$ is trivially also a refutation of $\overline{B} \wedge A$. In general, however, it does not hold that $\mathsf{Itp}(R, A, \overline{B})(\mathrm{s}_R) = \mathsf{Itp}(R, \overline{B}, A)(\mathrm{s}_R)$ (this follows immediately from Definition 3.3.3). Accordingly, it is possible to construct different interpolants from the same refutation R by merely changing the partitioning of the premises.

Example 3.3.6. *We split the formula presented in Example 3.3.5 into the partitions*

$$A \;\equiv\; \mathsf{e}_2 \wedge\; (\neg\mathsf{e}_2 \vee \mathtt{x}_1) \quad \textit{and} \quad \overline{B} \;\equiv\; \mathsf{e}_4 \wedge \mathsf{e}_5 \;\wedge\; (\neg\mathsf{e}_4 \vee \neg\mathtt{x}_1 \vee \mathsf{z}_0) \;\wedge\; (\neg\mathsf{e}_5 \vee \neg\mathsf{z}_0)\,.$$

Figure 3.7 shows the respective annotations for the proof in Figure 3.5. We obtain the propositional reverse interpolant $\mathtt{x}_1$, *which corresponds to the theory atom* $\mathtt{x}\&2 = 2$.

Remember that Lemma 3.2.2 establishes that the negation of a reverse interpolant of $(A, \overline{B})$ is a reverse interpolant for $(\overline{B}, A)$. The system $\mathsf{Itp}_{\mathrm{HKP}}$ has the property that

$\mathsf{Itp}(R, A, \overline{B})(v) = \neg\mathsf{Itp}(R, \overline{B}, A)(v)$ holds for all $v \in V_R$ ([Hua95], Lemma 13). D'Silva et al. [DPWK10] therefore refers to $\mathsf{Itp}_{\mathrm{HKP}}$ as "symmetric" system. McMillan's interpolation system, which we present next, does not have this property.

Definition 3.3.5 (McMillan's Interpolation System)**.** *The interpolation system Itp_M maps each vertex of a resolution refutation $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ of $A \wedge \overline{B}$ to a partial interpolant according to the following rules:*

1. *For each initial vertex v let*

$$\mathsf{Itp}_M(R, A, \overline{B})(v) \stackrel{\text{def}}{=} \begin{cases} \ell(v)|_B & \text{if } \ell(v) \in A \\ \textit{true} & \text{if } \ell(v) \in \overline{B} \end{cases} .$$

2. *Otherwise, v is an internal vertex with $(v^+, v) \in E_R$ and $(v^-, v) \in E_R$. Thus, $\ell(v)$ is the conclusion of a resolution step with the premises $\ell(v^+) = C \vee \mathtt{x}$ and $\ell(v^-) = D \vee \neg\mathtt{x}$ and the pivot $piv_R(v) = \mathtt{x}$. Then*

$$\mathsf{Itp}_M(R, A, \overline{B})(v) \stackrel{\text{def}}{=} \begin{cases} I_1 \vee I_2 & \text{if } \mathtt{x} \in \mathrm{Atoms}(A) \setminus \mathrm{Atoms}(B) \\ I_1 \wedge I_2 & \text{if } \mathtt{x} \in \mathrm{Atoms}(B) \end{cases} ,$$

   where

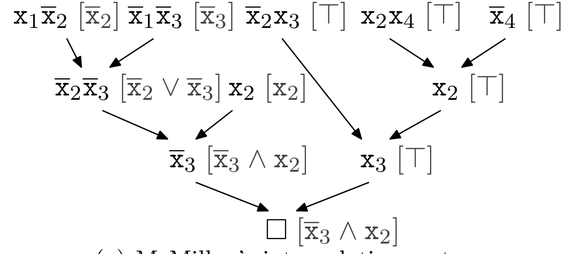
$$I_1 = \mathsf{Itp}_M(R, A, \overline{B})(v^+) \quad \textit{and} \quad I_2 = \mathsf{Itp}_M(R, A, \overline{B})(v^-) .$$

A proof of correctness is provided in [McM05]. In the following example, we apply $\mathsf{Itp}_{\mathrm{M}}$ to the refutation in Figure 3.5.

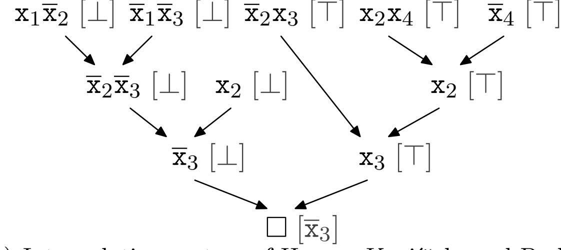
Example 3.3.7. *The annotations generated by the interpolation system Itp_M for the refutation in Figure 3.5 and the partitioning suggested in Example 3.3.5 differ from the annotations in Figure 3.6 only at two vertices. We obtain the annotations*

$$\overline{\mathsf{e}}_4\, \overline{\mathsf{x}}_1\, \mathsf{z}_0 \quad [\mathsf{z}_0] \quad \textit{and} \quad \overline{\mathsf{x}}_1\, \mathsf{z}_0 \quad [\mathsf{z}_0] .$$

All other annotations remain unchanged. In particular, $\mathsf{Itp}_M(R, A, \overline{B})(\mathrm{s}_R) = \mathsf{z}_0$, i.e, both interpolation systems yield the same reverse interpolant. (We demonstrate in Example 3.3.8



(a) McMillan's interpolation system



(b) Interpolation system of Huang, Krajíček, and Pudlák

Figure 3.8: Refutation yielding different interpolants for different systems

that this is not the case in general.) Using Lemma 3.2.1, we can show that there exists only a single interpolant for the formulae presented in Example 3.3.5. We obtain

$$\exists e_2, e_4, x_1 \,.\, e_2 \wedge e_4 \;\wedge\; (\neg e_2 \vee x_1) \;\wedge\; (\neg e_4 \vee \neg x_1 \vee z_0) \equiv z_0 \quad \textit{and}$$

$$\forall e_5 \,.\, \neg e_5 \;\vee\; (e_5 \wedge z_0) \equiv z_0 \,.$$

(by means of repeated application of the equivalence $\exists x \,.\, F(x) \equiv F(x)[x/\textit{true}] \vee F(x)[x/\textit{false}]$*, and similarly for universal quantification). Accordingly, the strongest and weakest interpolant coincide, ruling out the existence of any other logically non-equivalent interpolants.*

The following example shows that the interpolants obtained from $\mathsf{Itp}_{\mathrm{M}}$ and $\mathsf{Itp}_{\mathrm{HKP}}$ may differ and that $\mathsf{Itp}_{\mathrm{M}}$ does not have the symmetry property $\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B}) = \neg\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)$.

Example 3.3.8. *Let*

$$A \equiv (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge x_2 \quad \textit{and} \quad \overline{B} \equiv (\overline{x}_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge \overline{x}_4 \,.$$

An $(A, \overline{B})$*-refutation* R *is shown in Figure 3.8. The partial interpolants generated using McMillan's system are shown in Figure 3.8(a) and those generated using* Itp_{HKP} *in Figure 3.8(b). We obtain* $\mathsf{Itp}_M(R, A, \overline{B})(s_R) = x_2 \wedge \neg x_3$ *and* $\mathsf{Itp}_{HKP}(R, A, \overline{B})(s_R) = \neg x_3$*. If we*

swap the partitions, we obtain $\mathsf{Itp}_M(R, \overline{B}, A)(\mathrm{s}_R) = \mathtt{x}_2 \wedge \mathtt{x}_3$ *and* $\mathsf{Itp}_{HKP}(R, \overline{B}, A)(\mathrm{s}_R) = \mathtt{x}_3$.

The results obtained in Example 3.3.8 allow us to make several interesting observations:

1. The systems $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$ enable us to generate three different reverse interpolants, namely $\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})(\mathrm{s}_R) = \mathtt{x}_2 \wedge \neg\mathtt{x}_3$, $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(\mathrm{s}_R) = \neg\mathtt{x}_3$, and $\neg\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)(\mathrm{s}_R) = \neg\mathtt{x}_2 \vee \neg\mathtt{x}_3$.

2. It holds that $(\mathtt{x}_2 \wedge \neg\mathtt{x}_3) \Rightarrow \neg\mathtt{x}_3$ and $\neg\mathtt{x}_3 \Rightarrow (\neg\mathtt{x}_2 \vee \neg\mathtt{x}_3)$, i.e., $\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(\mathrm{s}_R)$ and $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \neg\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)(\mathrm{s}_R)$.

3. The uninterpreted predicate symbols in $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(\mathrm{s}_R)$ are a subset of the predicate symbols in $\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})(\mathrm{s}_R)$.

In the following, we show that these properties do not just hold by coincidence. We do so by generalising the interpolation systems $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$, i.e., we introduce a parametrised interpolation system $\mathsf{Itp}_{\mathrm{L}}$ which can be instantiated to a range of different interpolation systems including $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$. We show that these interpolation systems may yield interpolants of different logical strength.

Labelled Interpolation Systems

The last observation in the previous section (on page 101), though not related to the strength of interpolants, served as a catalyst for our work on parametrised interpolation systems [DPWK10]. Our colleague Mitra Purandare noticed that predicate symbols that are *peripheral* in the resolution proof need not be included in the interpolant.

Definition 3.3.6 (Peripherality [SDGC07])**.** *Let* $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ *be a resolution refutation of* $A \wedge \overline{B}$. *Given a node* $v \in V_R$ *and a literal* $l \in \{\mathtt{x}, \neg\mathtt{x}\}$ *such that* $l \in \ell(v)$, *the set of clauses* $S(l, v)$ *that "contribute"* l *is defined as follows:*

$$S(l,v) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } l \notin \ell(v) \\ \ell(v) & \text{if } v \text{ is an initial node and } l \in \ell(v) \\ S(l,v^+) \cup S(l,v^-) & \text{if } v \text{ is an internal node with } l \in \ell(v) \end{cases}$$

A literal $\mathtt{x}$ *is* peripheral *in* R *with respect to* A *and* $\overline{B}$ *if for every internal vertex* $v \in V_R$ *with* $piv_R(v) = \mathtt{x}$ *it holds that* $S(\mathtt{x}, v^+) \cup S(\neg\mathtt{x}, v^-) \subseteq A$ *or* $S(\mathtt{x}, v^+) \cup S(\neg\mathtt{x}, v^-) \subseteq \overline{B}$, *respectively.*

Note that a symbol $\mathtt{x}$ may be peripheral even if $S(\mathtt{x}, v_1^+) \cup S(\neg\mathtt{x}, v_1^-) \subseteq A$ and $S(\mathtt{x}, v_2^+) \cup S(\neg\mathtt{x}, v_2^-) \subseteq \overline{B}$ (where $piv(v_1) = piv(v_2) = \mathtt{x}$) for different *instances* of the literal $\mathtt{x}$. We are now in a position to formalise Purandare's observation.

Theorem 3.3.1 (Purandare's Peripherality Property). *Let* $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ *be a resolution refutation of* $A \wedge \overline{B}$ *and let* $X \subseteq \mathrm{Atoms}(A) \cap \mathrm{Atoms}(B)$ *be the set of atoms that are peripheral in* R*. Then, there exists an interpolant* I *for the pair of formulae* (A, B) *such that for all* $\mathtt{x} \in X$ *it holds that* $\mathtt{x} \notin \mathrm{Atoms}(I)$.

Later in this section, we show that by instantiating $\mathsf{Itp}_{\mathrm{L}}$ accordingly it is possible to obtain an interpolation system which yields interpolants satisfying the property stated in Theorem 3.3.1. We defer the respective proof to a point at which the definition of the parametrised interpolation system $\mathsf{Itp}_{\mathrm{L}}$ introduced in this section is available. We can, however, show a weaker version of Theorem 3.3.1 with the tools we have readily at hand.

Theorem 3.3.2 (Locality Property of $\mathsf{Itp}_{\mathrm{HKP}}$). *Let* $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ *be a resolution refutation of* $A \wedge \overline{B}$ *and let* $\mathtt{x} \in \mathrm{Atoms}(A) \cap \mathrm{Atoms}(B)$*. If for every inner vertex* $v \in V_R$ *with* $piv_R(v) = \mathtt{x}$ *and for all initial vertices* $u_1, \ldots, u_n$ *of the sub-graph with sink* v *it holds that either* $\bigcup_{i=1}^{n}\{\ell(u_i)\} \subseteq A$ *or* $\bigcup_{i=1}^{n}\{\ell(u_i)\} \subseteq B$*, then there is an interpolant* I *for* (A, B) *which is logically equivalent to* $\mathsf{Itp}_{HKP}(R, A, \overline{B})(\mathrm{s}_R)$ *such that* $\mathtt{x} \notin \mathrm{Atoms}(I)$ *holds.*

Proof. Given a resolution refutation $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ of $A \wedge \overline{B}$, we show that there exist formulae A' and $\overline{B}'$ such that $\mathtt{x} \notin \mathrm{Atoms}(A')$, $\mathtt{x} \notin \mathrm{Atoms}(\overline{B}')$, $A \vdash_{\mathcal{T}} A'$, and $\overline{B} \vdash_{\mathcal{T}} \overline{B}'$ are sub-proofs of R, and $A', \overline{B}' \vdash_{\mathcal{T}} \mathsf{false}$. Let I be a reverse interpolant for $(A', \overline{B}')$. Then it holds that $A' \Rightarrow I$ and $\overline{B}' \Rightarrow \neg I$. Since it holds that $\mathrm{Atoms}(A' \wedge \overline{B}') \subseteq \mathrm{Atoms}(A \wedge \overline{B})$, I is also a reverse interpolant for $(A, \overline{B})$. Furthermore, we show that $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(\mathrm{s}_R)$ is such a reverse interpolant for $(A', \overline{B}')$.

Let $v \in V_R$ be a vertex such that $piv_R(v) = \mathtt{x}$. Furthermore, assume that $u_1, \ldots, u_n$ are the initial vertices of the sub-graph $R^v = (V_{R^v}, E_{R^v}, piv_R, \ell_R, v)$ of R and that $\bigcup_{i=1}^{n}\{\ell(u_i)\} \in$

A. Then it holds that $\ell(u_1), \ldots, \ell(u_n) \vdash_{\mathcal{T}} \ell(v)$ and $\mathtt{x} \notin \ell(v)$. Therefore, by replacing the conjunction $\bigwedge_{i=1}^{n} \ell(u_i)$ in A with $\ell(v)$, we obtain a new formula in which the number of occurrences of $\mathtt{x}$ is strictly smaller than in A. Accordingly, we obtain A' by repeating this substitution for all $v \in V_R$ with $piv_R(v) = \mathtt{x}$. $\overline{B}'$ can be constructed analogously.

Let R' be the sub-graph of R which we obtain by replacing each respective sub-graph with sink v and $piv_R(v) = \mathtt{x}$ in R with a new initial vertex u such that $\ell_R(v) = \ell_{R'}(u)$. Note that R' is a refutation of $A' \wedge \overline{B}'$. According to Definition 3.3.4, $\mathsf{Itp}_{\mathrm{HKP}}(R', A', \overline{B}')(u)$ is false if $\ell_R(u)$ and true otherwise. It remains to show that $\mathsf{Itp}_{\mathrm{HKP}}(R', A', \overline{B}')(u) \equiv \mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(v)$.

To this end, we show by induction that $\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(v) \equiv \mathsf{false}$ if $\ell(v)$ is derived from clauses in A exclusively. Let u_i be an initial vertex of the sub-graph R^v with sink v. Then $\ell(u_i) \in A$ and therefore $\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(u_i) = \mathsf{false}$. The induction hypothesis is that for all $(u^-, u) \in E_{R^v}$ and $(u^+, u) \in E_{R^v}$ it holds that $\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(u^-) \equiv \mathsf{false}$ and $\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(u^+) \equiv \mathsf{false}$. According to Definition 3.3.4, we have to consider two cases:

- The pivot is local to A, i.e., $piv_{R^v}(u) \in \mathrm{Atoms}(A) \setminus \mathrm{Atoms}(B)$. Then it holds that $\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(u) = \mathsf{false} \vee \mathsf{false} \equiv \mathsf{false}$.
- Otherwise, the pivot $piv_{R^v}(u)$ is a shared symbol. Accordingly,

$$\mathsf{Itp}_{\mathrm{HKP}}(R^v, A, \overline{B})(u) = (\mathtt{x} \vee \mathsf{false}) \wedge (\neg\mathtt{x} \vee \mathsf{false}) = \mathtt{x} \wedge \neg\mathtt{x} \equiv \mathsf{false}\,.$$

This establishes that $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(v) \equiv \mathsf{false}$. Analogously, we can show that the partial interpolant $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(v)$ is logically equivalent to true if $\ell(v)$ is a fact which derives from clauses in $\overline{B}$ exclusively. This establishes that the annotations of the vertices of R' are logically equivalent to their respective counterparts in R. □

Theorem 3.3.2 states that we can treat a predicate symbol $\mathtt{x}$ shared by A and B as "local" if no literal over $\mathtt{x}$ contributed by A is ever resolved with a literal over $\mathtt{x}$ contributed by B. The following example provides an intuition for this mechanism.

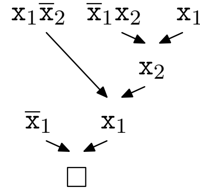


Figure 3.9: A resolution proof for $\neg x_1 \wedge (x_1 \vee \neg x_2)$ and $(\neg x_1 \vee x_2) \wedge x_1$

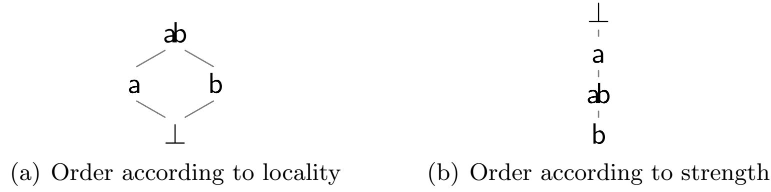
Example 3.3.9. *Consider the annotated refutation in Figure 3.8(b). Note that the clause* x_3 *is derived from* $\overline{B} \equiv (\neg x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge \neg x_4$ *exclusively by resolution. Resolution corresponds to existential quantification, i.e.,* $x_3 \equiv \exists x_2 \,.\, ((\neg x_2 \vee x_3) \wedge \exists x_4 \,.\, (x_2 \vee x_4) \wedge \neg x_4)$*. Intuitively, we can rename the quantified symbols to symbols which do not occur in A without changing the reverse interpolant. Accordingly,* x_2 *and* x_4 *need not occur in the interpolant.*

Peripherality, as introduced in Definition 3.3.6, is more general than the locality property observed Theorem 3.3.2. However, the mechanism used by $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$ to classify pivot elements as local or shared does not allow distinguishing between peripheral and non-peripheral symbols. Accordingly, the interpolation systems introduced in Definitions 3.3.4 and 3.3.5 do not allow us to take advantage of the peripherality property.

Example 3.3.10. *Consider the resolution refutation R in Figure 3.9 for the pair of propositional formulae* $A \equiv \neg x_1 \wedge (x_1 \vee \neg x_2)$ *and* $\overline{B} \equiv (\neg x_1 \vee x_2) \wedge x_1$*. We obtain the reverse interpolants* $\mathsf{Itp}_M(R, A, \overline{B}) = \mathsf{Itp}_{HKP}(R, A, \overline{B}) = \neg x_1 \wedge \neg x_2$ *and* $\neg\mathsf{Itp}_M(R, \overline{B}, A) = \neg x_1 \vee \neg x_2$*. Note that* x_1 *is peripheral in R with respect to A and $\overline{B}$ and, in accordance with Theorem 3.3.1, there is a reverse interpolant which does not contain this symbol, namely* $\neg x_2$*. However, it is not possible to obtain this interpolant by means of* Itp_M *or* Itp_{HKP}*.*

In order to identify peripheral symbols a more fine grained mechanism to classify literals is required. For this purpose, we introduce *labelling functions*, which allow us to tag individual literals as "shared" or "local to" A or $\overline{B}$.

Definition 3.3.7 (Labelling Function)**.** *Let* $(\mathcal{S}, \sqsubseteq, \sqcap, \sqcup)$ *be a lattice, where* $\mathcal{S} = \{\bot, \mathsf{a}, \mathsf{b}, \mathsf{ab}\}$ *is a set of labels and* $\sqsubseteq$*,* $\sqcap$ *and* $\sqcup$ *are defined by the Hasse diagram in Figure 3.10(a). A* labelling function *L for a refutation R is a total mapping from tuples of literals l and vertices*



(a) Order according to locality (b) Order according to strength

Figure 3.10: Different orders for $\{\bot, \mathsf{a}, \mathsf{b}, \mathsf{ab}\}$

$v \in V_R$ to elements in $\mathcal{S}$ such that the following condition holds:

$$L(v, l) = \begin{cases} \bot & \textit{if and only if } l \notin \ell_R(v) \\ L(v^+, l) \sqcup L(v^-, l) & \textit{for an internal vertex } v \textit{ and a literal } l \in \ell_R(v) \end{cases}$$

The condition in Definition 3.3.7 implies that the labelling function for literals at internal vertices is completely determined by the labels of literals at initial vertices. It does, however, impose no restrictions on the labels of literals at the initial vertices. Intuitively, these labels are an indicator of the "locality" of the literals with respect to the formulae A and $\overline{B}$. A propositional symbol $\mathtt{x}$ is local to A (or A-*local*) in a pair $(A, \overline{B})$ if $\mathtt{x} \in \text{Atoms}(A) \setminus \text{Atoms}(\overline{B})$, local to $\overline{B}$ (or $\overline{B}$-*local*) if $\mathtt{x} \in \text{Atoms}(\overline{B}) \setminus \text{Atoms}(A)$, *local* if it is either of these, and *shared* otherwise. In the following Definition 3.3.8, we introduce restrictions for labelling functions that are consistent with this notion of locality (which is also used in Definitions 3.3.4 and 3.3.5) but liberal enough to enable the definition of peripheral symbols (Definition 3.3.6) on the basis of labelling functions.

Definition 3.3.8 (Locality)**.** *A labelling function for an $(A, \overline{B})$-refutation R* preserves locality *if for any initial vertex v and literal l in R*

1. $\mathsf{a} \sqsubseteq L(v, l)$ *implies that* $\text{Atoms}(l) \subseteq \text{Atoms}(A)$*, and*

2. $\mathsf{b} \sqsubseteq L(v, l)$ *implies that* $\text{Atoms}(l) \subseteq \text{Atoms}(\overline{B})$ *.*

The labelling function introduced in [SDGC07] is an example of a locality-preserving labelling function.

Definition 3.3.9 (Labelling Function for Peripherality [SDGC07])**.** *L_P is a locality-preserving labelling function for a $(A, \overline{B})$-refutation R such that the following condition holds:*

$$v \in V_R \text{ is an initial vertex} \Rightarrow L_P(v, l) = \begin{cases} \bot & \text{iff } l \notin \ell_R(v) \\ \mathsf{a} & \text{iff } l \in \ell_R(v) \text{ and } \ell_R(v) \in A \\ \mathsf{b} & \text{iff } l \in \ell_R(v) \text{ and } \ell_R(v) \in \overline{B} \end{cases}$$

It follows immediately from the Definitions 3.3.6 and 3.3.7 that a symbol $\mathtt{x}$ is peripheral in a proof R if (and only if) for every internal vertex $v \in V_R$ with $piv(v) = \mathtt{x}$ it holds that $L_P(v^+, \mathtt{x}) = L_P(v^-, \neg\mathtt{x}) = \mathsf{a}$ or $L_P(v^+, \mathtt{x}) = L_P(v^-, \neg\mathtt{x}) = \mathsf{b}$. Accordingly, the labelling function L_P represents an algorithm for detecting peripherality [SDGC07]. We emphasise that [SDGC07] does not use the labelling function L_P for the purpose of interpolant generation, and that our definition of labelling functions is more general.

Analogously to the projection $C|_A$ (see Definition 3.2.3) we define the downward and upward projection for labelled literals.

Definition 3.3.10 (Upward and Downward Projection)**.** *Given a labelling function L, the downward and upward* projection *of a clause at a vertex v with respect to $\mathsf{c} \in \mathcal{S}$ is defined as*

$$\ell(v)\lfloor_{\mathsf{c},L} \stackrel{\text{def}}{=} \{l \in \ell(v) \mid L(v, l) \sqsubseteq \mathsf{c}\} \quad \textit{and} \quad \ell(v)\lceil_{\mathsf{c},L} \stackrel{\text{def}}{=} \{l \in \ell(v) \mid \mathsf{c} \sqsubseteq L(v, l)\}\,,$$

respectively.

We drop the subscript L if the labelling function is clear from the context. We proceed to define a novel interpolation system based on labelling functions.

Definition 3.3.11 (Labelled Interpolation System)**.** *Let L be a locality preserving labelling function for an $(A, \overline{B})$-refutation R. The labelled interpolation system Itp_L maps vertices in R to partial interpolants as defined below.*

1. *For each initial vertex v let*

$$\mathsf{Itp}_L(R, A, \overline{B})(v) \stackrel{\text{def}}{=} \begin{cases} \ell(v)\lfloor_{\mathsf{b}} & \text{if } \ell(v) \in A \\ \neg(\ell(v)\lfloor_{\mathsf{a}}) & \text{if } \ell(v) \in \overline{B} \end{cases}.$$

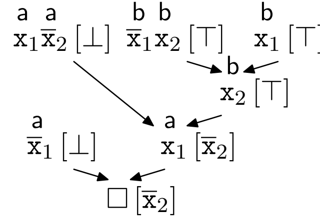


Figure 3.11: An annotated refutation obtained using a labelled interpolation system $\mathsf{Itp}(L)$

2. Otherwise, v is an internal vertex with $(v^+, v) \in E_R$ and $(v^-, v) \in E_R$. Thus, $\ell(v)$ is the conclusion of a resolution step with the premises $\ell(v^+) = C \vee \mathtt{x}$ and $\ell(v^-) = D \vee \neg\mathtt{x}$ and the pivot $piv_R(v) = \mathtt{x}$. Then

$$\mathsf{Itp}_L(R, A, B)(v) \stackrel{\text{def}}{=} \begin{cases} I_1 \vee I_2 & \text{if } L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{a} \\ (\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2) & \text{if } L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{ab} \\ I_1 \wedge I_2 & \text{if } L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{b} \end{cases},$$

where

$$I_1 = \mathsf{Itp}_L(R, A, \overline{B})(v^+) \quad \text{and} \quad I_2 = \mathsf{Itp}_L(R, A, \overline{B})(v^-)\,.$$

Note that L is a parameter of the interpolation system. Whenever we want to emphasise this parametrisation, we write $\mathsf{Itp}(L, R, A, \overline{B})$ for the interpolant obtained from an $(A, \overline{B})$-refutation R with a labelling function L. Again, we omit the parameters A and $\overline{B}$ whenever they are clear from the context. Example 3.3.11 illustrates the use of a labelled interpolation system based on the labelling function L_P (see Definition 3.3.9). Our claim that labelled interpolation systems are strictly more general than existing systems is substantiated by constructing an interpolant that cannot be obtained from $\mathsf{Itp}_{\mathrm{M}}$ and $\mathsf{Itp}_{\mathrm{HKP}}$.

Example 3.3.11. *As in Example 3.3.10, let $A \equiv \neg\mathtt{x}_1 \wedge (\mathtt{x}_1 \vee \neg\mathtt{x}_2)$ and $\overline{B} \equiv (\neg\mathtt{x}_1 \vee \mathtt{x}_2) \wedge \mathtt{x}_1$. A corresponding refutation of $A \wedge \overline{B}$ is shown in Figure 3.11 with the symbol $L_P(v, l)$ above each literal l. The interpolant obtained from $\mathsf{Itp}(L_P, R, A, \overline{B})$ is $\neg\mathtt{x}_2$. Recall from Example 3.3.10 that this interpolant cannot be derived in existing systems.*

It remains to show that the labelled interpolation system introduced in Definition 3.3.11 does indeed generate valid interpolants and that it is capable of simulating existing interpolation systems.

Theorem 3.3.3 (Correctness)**.** *For any resolution refutation R of a formula $A \wedge \overline{B}$ and locality preserving labelling function L, $\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R)$ is a reverse interpolant for $(A, \overline{B})$.*

We prove the theorem by showing that the annotation $I = \mathsf{Itp}(L, R, A, \overline{B})(v)$ satisfies the following conditions for all $v \in V_R$:

1. $A \wedge \neg(\ell(v)\upharpoonright_{\mathsf{a},L}) \Rightarrow I$,
2. $\overline{B} \wedge \neg(\ell(v)\upharpoonright_{\mathsf{b},L}) \Rightarrow \neg I$, and
3. $\mathrm{Atoms}(I) \subseteq \mathrm{Atoms}(A) \cap \mathrm{Atoms}(\overline{B})$.

These conditions resemble the invariants $A \wedge \neg(\ell(v)|_A) \Rightarrow I$ and $\overline{B} \wedge \neg(\ell(v)|_B) \Rightarrow \neg I$ satisfied by the interpolation system $\mathsf{Itp}_{\mathrm{HKP}}$, which warrant that each annotation generated by $\mathsf{Itp}_{\mathrm{HKP}}$ is a partial interpolant according to Definition 3.2.4. In order to accommodate the interpolation systems $\mathsf{Itp}_{\mathrm{M}}$ and $\mathsf{Itp}_{\mathrm{L}}$, we adapt the definition of partial interpolants as follows:

Definition 3.3.12 (Partial Interpolant for Resolution Refutations)**.** *Let R be a resolution refutation $(V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ of $A \wedge \overline{B}$, a propositional formula in CNF. Furthermore, let L be a locality preserving labelling function for R. A* partial interpolant *for a vertex $v \in V_R$ is a formula I for which the following conditions hold*

1. $A \wedge \neg(\ell(v)\upharpoonright_{\mathsf{a},L}) \Rightarrow I$,

2. $I \Rightarrow B \vee (\ell(v)\upharpoonright_{\mathsf{b},L})$, *and*

3. $\mathrm{Atoms}(I) \subseteq \mathrm{Atoms}(A) \cap \mathrm{Atoms}(B)$.

Since R is a refutation, it holds that $\ell(\mathrm{s}_R) = \square$. Therefore, the partial interpolant for s_R is a Craig interpolant for the pair of formulae (A, B).

We obtain this specialised definition of partial interpolants by simply replacing the projection operator (Definition 3.2.3) in Definition 3.2.4 with the upward projection operator (Definition 3.3.10). Similarly, apart from a minor adaption to accommodate labelling functions, our proof of correctness resembles the proof provided by Yorsh and Musuvathi

in [YM05]. Our proof by induction is provided in Section B.2 of Appendix B and in our technical report [DKPW09].

Labelled interpolation systems do not only enable the generation of interpolants which cannot be obtained by means of $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$ (as demonstrated in Example 3.3.11), they also allow us simulate existing interpolation systems. The following Lemma shows that $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$ are instances of the interpolation system introduced in Definition 3.3.11.

Since labelling functions are completely determined by the labels at the initial vertices of R (cf. Definition 3.3.7), it suffices to define the labelling functions which yield $\mathsf{Itp}_{\mathrm{HKP}}$ and $\mathsf{Itp}_{\mathrm{M}}$ at the initial vertices.

Lemma 3.3.1. *Let R be a resolution refutation for the formula $A \wedge \overline{B}$. The labelling functions L_{HKP}, L_M and $L_{M'}$ are defined for initial vertices v and literals $l \in \ell(v)$ as follows:*

Atoms(l)	$L_M(v,l)$	$L_{HKP}(v,l)$	$L_{M'}(v,l)$
A-local	a	a	a
shared	b	ab	a
$\overline{B}$-local	b	b	b

The following equalities hold for all $v \in V_R$:

$$\begin{aligned} \mathsf{Itp}_M(R, A, \overline{B})(v) &= \mathsf{Itp}(L_M, R, A, \overline{B})(v) \\ \mathsf{Itp}_{HKP}(R, A, \overline{B})(v) &= \mathsf{Itp}(L_{HKP}, R, A, \overline{B})(v) \\ \neg\mathsf{Itp}_M(R, \overline{B}, A)(v) &= \mathsf{Itp}(L_{M'}, R, A, \overline{B})(v) \end{aligned}$$

We provide a proof of this claim in Section B.2. In Lemma 3.3.1, the label of each literal at an initial vertex is determined exclusively by whether the corresponding symbol is A-local, $\overline{B}$-local or shared. In general, any conceivable locality-preserving labelling function is allowed. The labelling function L_P in Definition 3.3.9 also determines the label based on the locality of symbols, but may assign different symbols to different occurrences of the same literal ($\mathtt{x}_1$ in Figure 3.11, for instance).

The interpolation system $\mathsf{Itp}_{\mathrm{L}}$ in combination with the labelling function L_P (Definition 3.3.9, applied in Example 3.3.11) establishes the correctness of Theorem 3.3.1. Given

a symbol $\mathtt{x}$ that is peripheral in R it holds for all nodes $v \in V_R$ that either

- v is an initial node and if $\ell(v) \in A$ with $l \in \ell(v)$ and $l \in \{\mathtt{x}, \neg\mathtt{x}\}$ then $L_P(v,l) = \mathsf{a}$ (or b if $\ell(v) \in \overline{B}$, respectively), or
- v is an internal node with predecessors v^+ and v^- and $L_P(v^+, \mathtt{x}) \sqcup L_P(v^-, \neg\mathtt{x}) \neq \mathsf{ab}$.

Then, according to Definition 3.3.11, the annotations of the initial nodes do not contain the symbol $\mathtt{x}$. Furthermore, $\mathtt{x}$ is also not introduced at the inner nodes v with $piv(v) = \mathtt{x}$. It follows that $\mathtt{x} \notin \mathsf{Itp}(L_P, R, A, \overline{B})(\mathrm{s}_R)$.

Evidently, labelled interpolation systems (Definition 3.3.11) in combination with Theorem 3.3.1 provide us with a mechanism to eliminate symbols that are peripheral in a refutation R for $A \wedge \overline{B}$ from the interpolant for the formulae (A, B). Furthermore, for a fixed resolution refutation R and *any* locality-preserving labelling function L it holds that $\mathrm{Atoms}(\mathsf{Itp}(L_P, R, A, \overline{B})(\mathrm{s}_R)) \subseteq \mathrm{Atoms}(\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R))$. Moreover, the set-inclusion order for the symbols occurring in the interpolants $\mathsf{Itp}(L_1, R, A, \overline{B})(\mathrm{s}_R)$ and $\mathsf{Itp}(L_2, R, A, \overline{B})(\mathrm{s}_R)$ also imposes an order on the labelling functions L_1 and L_2. A proof and a formal discussion of this matter is provided in [D'S10]. On a related note, an interpolation technique which is based on binary decision diagrams and enables the exclusion of predicate symbols by means of quantifier elimination is presented in [EKS06]. In the following section, we focus our attention on the implication order and the logical strength of interpolants.

3.3.4 Modulating Interpolant Strength

Recall that on page 101 we observe that different interpolation systems enable us to generate interpolants of varying logical strength (as demonstrated in Example 3.3.8). We set out to show that labelled interpolation systems enable us to systematically tune the strength of interpolants. In this section, we show that the strength of an interpolant obtained by means of $\mathsf{Itp}(L, R, A, \overline{B})$ is directly determined by the labelling function L.

The labels of the pivot elements in R determine how the interpolation system $\mathsf{Itp}_{\mathrm{L}}$ combines partial interpolants. Let us review Definition 3.3.11. At each inner node, $\mathsf{Itp}_{\mathrm{L}}$ introduces a disjunction $(I_1 \vee I_2)$, multiplexer $(\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2)$, or conjunction $(I_1 \vee I_2)$ over the preceding partial interpolants, depending on the label of the pivot element $\mathtt{x}$. Figure 3.12

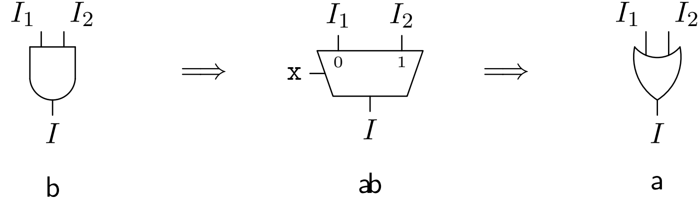


Figure 3.12: Combining partial interpolants: $(I_1 \wedge I_2) \Rightarrow (\mathtt{x} \vee I_1) \wedge (I_2 \vee \neg \mathtt{x}) \Rightarrow (I_1 \vee I_2)$

visualises this case split and illustrates the crucial observation that the resulting partial interpolants are ordered by strength. Pivot elements labelled b result in stronger partial interpolants than pivot elements labelled ab, which in turn yield stronger annotations than pivots labelled a. This observation gives rise to the ordering relation $\preceq$ defined by the Hasse diagram in Figure 3.10(b). Note that the total order $\preceq$ differs from the order $\sqsubseteq$ introduced in Definition 3.3.7 (see Figure 3.10(a)). By abuse of notation, we also use $\preceq$ to denote the point-wise extension of $\preceq$ to labelling functions. Formally, we define the *strength order* $\preceq$ as follows:

Definition 3.3.13 (Strength Order)**.** *Let $\preceq$ denote the total order on $\mathcal{S} = \{\bot, \mathsf{a}, \mathsf{b}, \mathsf{ab}\}$ such that $\mathsf{b} \preceq \mathsf{ab} \preceq \mathsf{a} \preceq \bot$ holds (cf. Figure 3.10(b)). Given two labelling functions L and L' for a resolution refutation R of $A \wedge \overline{B}$, we say that the function L is stronger than L', denoted $L \preceq L'$, if for all $v \in V_R$ and $l \in \ell(v)$, $L(v,l) \preceq L'(v,l)$.*

Note that $\preceq$ is a partial order on labelling functions. According to Definition 3.3.7, the labelling function for the initial nodes of a resolution refutation determines its values for all inner vertices. Conveniently, a similar claim holds for the strength of labelling functions.

Lemma 3.3.2. *Let L and L' be labelling functions for a resolution refutation R for $A \wedge \overline{B}$. If $L(v,l) \preceq L'(v,l)$ for all initial vertices v and literals $l \in \ell(v)$, then $L \preceq L'$.*

The following theorem states that there is a direct correspondence between the strength of a labelling function L and the resulting interpolation system $\mathsf{Itp}(L, R, A, \overline{B})$. Naturally, a labelled interpolation system $\mathsf{Itp}(L, R, A, \overline{B})$ is *stronger than* $\mathsf{Itp}(L', R, A, \overline{B})$ if for all resolution refutations R of $A \wedge \overline{B}$, $\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L', R, A, \overline{B})(\mathrm{s}_R)$.

Theorem 3.3.4. *Let L and L' be labelling functions for an refutation R of the formula $A \wedge \overline{B}$. If $L \preceq L'$, then $\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L', R, A, \overline{B})(\mathrm{s}_R)$.*

An intuitive explanation for this claim is provided at the beginning of the section. Formally, the correctness of Theorem 3.3.4 follows from the fact that for each vertex $v \in V_R$ the following invariant holds:

$$(L \preceq L') \quad \Rightarrow \quad \big(\mathsf{Itp}(L, R, A, \overline{B})(v) \Rightarrow \mathsf{Itp}(L', R, A, \overline{B})(v) \vee (\ell_R(v)|_A \cap \ell_R(v)|_B)\big)$$

For $\ell(\mathrm{s}_R) = \square$, this invariant establishes the correctness of the theorem. We provide the complete proofs for Lemma 3.3.2 and Theorem 3.3.4 in Section B.3.

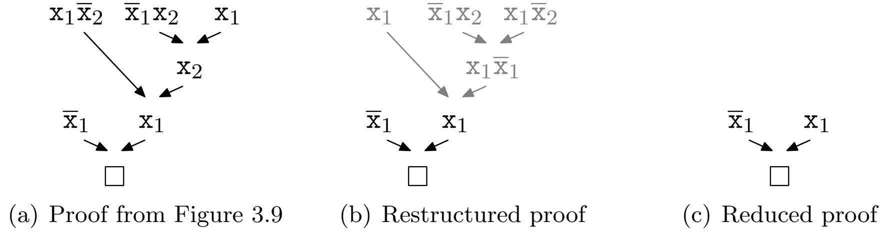
Consequently, labelling functions do not only provide us with a mechanism to exclude symbols that are peripheral in the resolution proof, they also enable us to strengthen and weaken the interpolants generated using labelled interpolation systems. The strength of an interpolant can be varied by simply strengthening or weakening the underlying labelling function. Furthermore, analogously to the combination of interpolants in Lemma 3.2.3, we can combine labelling functions.

Definition 3.3.14. *We use* $\max(\mathsf{c}_1, \mathsf{c}_2)$ *and* $\min(\mathsf{c}_1, \mathsf{c}_2)$ *to denote the maximum and minimum, respectively, of the symbols* $\mathsf{c}_1, \mathsf{c}_2 \in \mathcal{S}$ *(with respect to* $\preceq$*). Let* R *be a resolution refutation and* L_1 *and* L_2 *be labelling functions. The labelling functions* $L_1 \Uparrow L_2$ *and* $L_1 \Downarrow L_2$ *are defined for any initial vertex* v *and literal* $l \in \ell(v)$ *as follows:*

- $(L_1 \Uparrow L_2)(v, l) = \max(L_1(v, t), L_2(v, l))$*, and*
- $(L_1 \Downarrow L_2)(v, l) = \min(L_1(v, t), L_2(v, l))$*.*

The label of an internal vertex v *and* $l \in \ell(v)$*, is defined inductively as usual.*

Lemma 3.2.1 provides a general upper and lower bound for the strength of an interpolant. The interpolations systems discussed in this section, however, do not necessarily enable us to compute the strongest and weakest interpolant. It turns out that the set of labelling functions ordered by $\preceq$ has L_M and $L_{M'}$ as the least and greatest element, respectively. Accordingly, McMillan's interpolation system defines the lower and upper bound for the strength of an interpolant generated using labelled interpolation systems. This observation is formalised in the following theorem. A proof can be found in Section B.3.



(a) Proof from Figure 3.9 (b) Restructured proof (c) Reduced proof

Figure 3.13: Different resolution refutations for $\neg x_1 \wedge (x_1 \vee \neg x_2)$ and $(\neg x_1 \vee x_2) \wedge x_1$

Theorem 3.3.5. *Let R be a refutation of $A \wedge \overline{B}$ and $\mathbb{L}_R$ be the set of locality preserving labelling functions over R. The structure $(\mathbb{L}_R, \preceq, \Uparrow, \Downarrow)$ is a complete lattice with L_M as the least and $L_{M'}$ as the greatest element.*

3.3.5 Proof Transformations and Interpolant Strength

As indicated in Section 3.2, the interpolant generated by an interpolation system is not only determined by the labelling function but also by the structure of the underlying proof. In fact, there may be valid interpolants which cannot be derived from a certain refutation. The following example shows that different proofs enable the generation of different interpolants.

Example 3.3.12. *Recall from Example 3.3.10 that $\neg x_2 \wedge \neg x_1$, $\neg x_2 \vee \neg x_1$, $\neg x_2$, and $\neg x_1$ are reverse interpolants for the pair of formulae $\neg x_1 \wedge (x_1 \vee \neg x_2)$ and $(\neg x_1 \vee x_2) \wedge x_1$. The former three interpolants can be derived from the proof R in Figure 3.9 (Figure 3.13(a), respectively) using the interpolation systems Itp_M and $\mathsf{Itp}(L_P)$ (see Example 3.3.11). This is not the case for the formula $\neg x_1$, however. To see this, note that the proof in Figure 3.9 contains a vertex v with $piv(v) = x_2$. The literals x_2 and $\neg x_2$ which are involved in this resolution step derive from different partitions. An analysis of the cases in Definition 3.3.11 tells us that any interpolant derived from R using a labelled interpolation system will contain the atom x_2. Note, however, that Itp_M generates the annotated proof*

$$\dfrac{\neg x_1 \quad [\neg x_1] \qquad\qquad x_1 \quad [\mathit{true}]}{\square \quad [\neg x_1]}\ \mathsf{Res}$$

for the refutation in Figure 3.13(c) and thus yields the reverse interpolant $\neg x_1$. Moreover, it is obvious that we cannot use labelled interpolation systems to extract the interpolants

$\neg x_2 \wedge \neg x_1$ *and* $\neg x_2 \vee \neg x_1$ *from the refutation in Figure 3.13(c).*

The refutation proofs in Example 3.3.12 are chosen in an *ad hoc* manner. In this section, we discuss techniques to systematically restructure a given refutation proof and demonstrate the impact of such modifications on the strength of the respective interpolants. We provide a formal analysis of the proof transformations presented by Jhala and McMillan in [JM07]. In the setting presented in [JM07], where transition functions are approximated using interpolants, weaker interpolants can slow down the convergence of the verification process. [JM07] shows that interpolants can be strengthened (or weakened) by changing the order of resolution steps in a refutation. Example 3.3.13 illustrates such a transformation.

Example 3.3.13. *We continue working in the setting of Example 3.3.8. Again, consider the formulae*

$$A \equiv (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge x_2 \quad \textit{and}$$

$$\overline{B} \equiv (\neg x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge \neg x_4$$

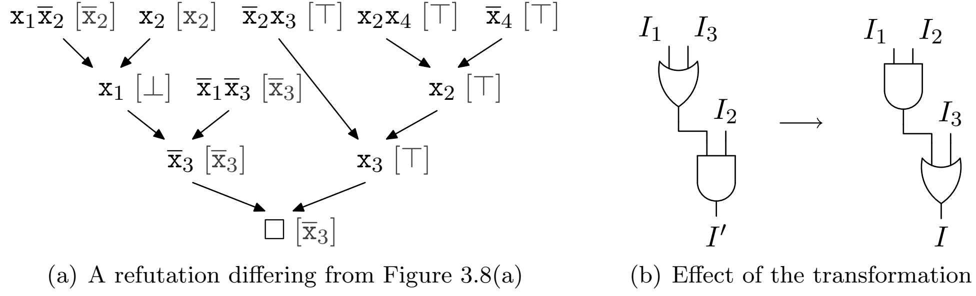
and the respective refutation R_1 *in Figure 3.8(a) (as previously discussed in Example 3.3.8). Figure 3.14(a) shows an alternative refutation* R_2*. The interpolant* $\mathsf{Itp}_M(R_1, A, \overline{B})(s_{R_1})$ *is* $x_2 \wedge \neg x_3$ *and* $\mathsf{Itp}_M(R_2, A, \overline{B})(s_{R_2})$ *is* $\neg x_3$*. Observe that* $x_2 \wedge \neg x_3$ *implies* $\neg x_3$*.*

The same interpolation system Itp_M *yields different interpolants for the refutations* R_1 *(Figure 3.8(a)) and* R_2 *(Figure 3.14(a)). The refutation* R_1 *differs from* R_2 *in so far as the order of the resolution steps of the clause* $x_1 \vee \neg x_2$ *with the clauses* x_2 *and* $\neg x_1 \vee \neg x_3$ *(in the leftmost branch of the proof) is reversed. Since* x_1 *is local to* A *and* x_2 *is shared, this also changes the order in which the Boolean connectives* $\wedge$ *and* $\vee$ *are introduced in the partial interpolants. The effect of the transformation is illustrated in Figure 3.14(b), where partial interpolants are viewed as circuits. Since*

$$((I_1 \vee I_2) \wedge I_3) \quad \equiv \quad (I_1 \wedge I_3) \vee (I_2 \wedge I_3) \quad \textit{and} \quad (I_1 \wedge I_3) \vee (I_2 \wedge I_3) \quad \Rightarrow \quad ((I_1 \wedge I_2) \vee I_3)$$

we conclude that the transformation strengthens the partial interpolant at the vertex labelled $\overline{x}_3$ *in the Figures 3.8(a) and 3.14(a).*

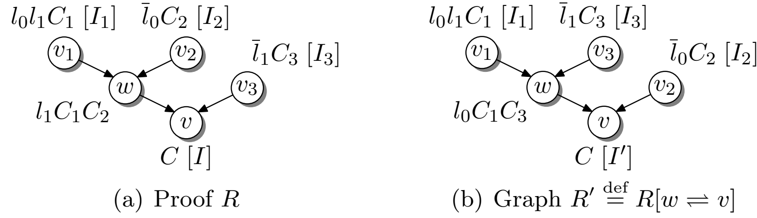
Intuitively, if resolutions on A-local symbols precede those on shared or $\overline{B}$-local symbols,



(a) A refutation differing from Figure 3.8(a)

(b) Effect of the transformation

Figure 3.14: A refutation obtained by swapping vertices in Figure 3.8(a)



(a) Proof R

(b) Graph $R' \stackrel{\text{def}}{=} R[w \rightleftharpoons v]$

Figure 3.15: Illustrations of refutations accompanying Definition 3.3.15

the interpolant is closer to CNF, hence more constrained and stronger. Naturally, the converse holds if the order is reversed.

In the following, we formalise the transformation illustrated in Example 3.3.13. Definition 3.3.15 introduces a "swap" transformation for vertices (and resolution steps, respectively) in resolution refutations. In order to simplify the presentation, we assume the proof to be tree shaped. The definition is based on the sub-graph presented in Figure 3.15(a). We assume that v and w are the vertices to be swapped and that v_1, v_2 and v_3 are their ancestors. The vertices are annotated with the respective clauses and the corresponding partial interpolants.

Definition 3.3.15 (Swap)**.** *Let $R = (V_R, E_R, piv, \ell_R, \mathrm{s}_R)$ be a tree-shaped refutation of $A \wedge \overline{B}$ and let v_1, v_2, v_3, v and w be vertices in V_R. We assume w.l.o.g. that the clauses and partial interpolants at these vertices and the connecting edges are as shown in Figure 3.15(a). Figure 3.15(b) shows the graph $R' = (V', E', piv', \ell', \mathrm{s}')$, which is the result of swapping the vertices w and v (denoted $R[w \rightleftharpoons v]$). Formally*

$$V' \stackrel{\text{def}}{=} V_R \qquad \textit{and} \qquad E' \stackrel{\text{def}}{=} (E \setminus \{(v_2, w), (v_3, v)\}) \cup \{(v_3, w), (v_2, v)\}\,.$$

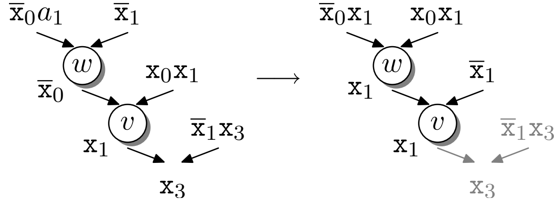


Figure 3.16: The graph $R[w \rightleftharpoons v]$ is not a proof because $\mathtt{x}_1$ is merged.

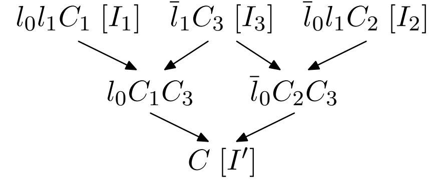


Figure 3.17: Transformation for the case where l_0 is a merge literal

The desired effect of the swapping operation is that $piv'(w) \stackrel{\text{def}}{=} piv_R(v)$ *and* $piv'(v) = piv(w)$ *and for all* $u \in V' \setminus \{v, w\}$*, the pivot function remains unchanged (i.e.,* $piv'(u) \stackrel{\text{def}}{=} piv_R(u)$*). Furthermore, for all vertices* $u \in V'$*,* $\ell'(u) \stackrel{\text{def}}{=} \ell_R(u)$ *if* $u \neq w$ *and* $\ell'(w) \stackrel{\text{def}}{=} t_0 \vee C_1 \vee C_3$*.*

The swap transformation does not change the set of vertices and the labels of all vertices except w remain unchanged. Unfortunately, in certain cases this transformation has the undesired side-effect that it yields an invalid resolution proof. The reason is that $\ell'(v)$ may not be the resolvent of $\ell(v^+)$ and $\ell(v^-)$, as demonstrated in the following example.

Example 3.3.14. *Consider the transformation illustrated in Figure 3.16. Note that the resolution on* $\neg\mathtt{x}_0$ *and* $\mathtt{x}_0 \vee \mathtt{x}_1$ *with* $piv(v) = \mathtt{x}_0$ *reintroduces the previously eliminated literal* $\mathtt{x}_1$ *in the proof on the left side. The reversal of the order of the nodes* v *and* w *results in an invalid resolution proof, since the literal* $\mathtt{x}_1$ *occurring in the clauses* $\neg\mathtt{x}_0 \vee \mathtt{x}_1$ *and* $\mathtt{x}_0 \vee \mathtt{x}_1$ *is* merged *in the resolution at* w *with* $piv(w) = \mathtt{x}_0$*, making the subsequent resolution on* $\mathtt{x}_1$ *superfluous.*

Example 3.3.14 is a counterexample to the claim in [JM07, page 11] that the swap transformation $R[w \rightleftharpoons v]$ "*is valid when* q *occurs in* v_1*, but not in* v_2" (where q denotes $piv(w)$). Andrews [And68] uses the term *merge literal* to refer to a literal such as $\mathtt{x}_1$ in Example 3.3.14. Formally, a merge literal is a literal $l \in \ell(v)$ such that $l \in \ell(v^+) \cap \ell(v^-)$, where $(v^-, v) \in E_R$ and $(v^+, v) \in E_R$. Given a proof R as in Figure 3.15(a), the presence of merge literals leads to an invalid label $\ell'(v)$ in the transformed proof $R[w \rightleftharpoons v]$ (see Figure 3.15(b) and Definition 3.3.15)

- if $l_1 \in C_2$, since in that case l_1 does not occur in C but in the resolvent of $\ell'(w)$ and $\ell'(v_2)$, or

- if $l_0 \in C_3$, since then l_0 occurs in C but not in the resolvent of $\ell'(w)$ and $\ell'(v_2)$.

In both cases, the $\ell'(v)$ is not the resolvent of its antecedents. (We address both of these two cases below.) For all remaining cases, we say that the edge (w, v) is *merge-free*. Formally, given a proof R with vertices v and w connected and labelled as in Figure 3.15(a), an edge (w, v) in R is *merge-free* if $l_0 \notin \ell_R(v_3)$ and $l_1 \notin \ell_R(v_2)$. Lemma 3.3.3 shows that $R[w \rightleftharpoons v]$ is a valid proof for merge-free edges (w, v). The proof of the lemma can be found in Section B.4.

Lemma 3.3.3. *Let R be a proof with vertices v and w connected and labelled as in Figure 3.15(a). If (w, v) is merge-free, then $R[w \rightleftharpoons v]$ is a resolution proof.*

The first of the problematic cases mentioned above is taken care of in [JM07] by adding an additional resolution for the clauses $(\neg l_1 \vee C_3)$ and $(\neg l_0 \vee C_2)$ in order to eliminate l_1 from $\ell'(v)$. The resulting graph is shown in Figure 3.17. Observe that in both branches the order of the resolution steps is reversed with respect to the proof in Figure 3.15(a). This transformation has the interesting property that it does not change the annotations generated by $\mathsf{Itp}_{\mathrm{M}}$ and $\mathsf{Itp}_{\mathrm{HKP}}$ for v. In the following, we demonstrate this for two cases. A formal proof is provided in Section B.4 (see Lemma B.4.1).

1. If l_1 is local to A and l_0 is shared, the partial interpolants I and I', shown as circuits in Figure 3.18(a), are $I = (I_1 \wedge I_2) \vee I_3$ and $I' = (I_1 \vee I_3) \wedge (I_2 \vee I_3)$, respectively. Intuitively, the transformation distributes the disjunction.

2. Now assume that l_0 is local to B and l_1 is shared. In this case, $\mathsf{Itp}_{\mathrm{M}}$ introduces a conjunction at both vertices. Furthermore, the circuits in Figure 3.18(b) show that this transformation does not change the interpolants in $\mathsf{Itp}_{\mathrm{HKP}}$ in this case.

Example 3.3.14 represents a case in which $l_0 \in C_3$ in Figure 3.15(a). In this case, the transformation yields a proof $R[w \rightleftharpoons v]$ in which l_0 has been eliminated from $\ell'(v)$. Accordingly, $R[w \rightleftharpoons v]$ contains a superfluous resolution on l_0. The elimination of the superfluous resolution may eliminate additional literals from the respective resolvent, creating in an avalanche effect which results in a significant reduction of the size of the proof. A algorithm which performs such a repair is presented in [BIFH$^+$09]. An alternative approach to avoid

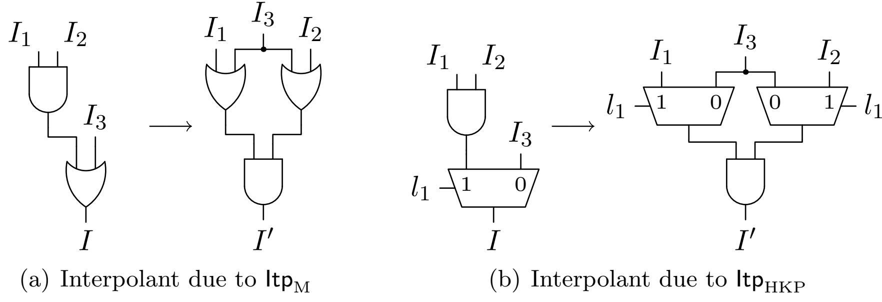


Figure 3.18: Transforming R in Figure 3.15(a) as in Figure 3.17 does not change the interpolant.

the problem is to use multi-sets to represent clauses. This approach, however, requires an additional factoring rule which necessitates the adaptation of the labelled interpolation system in Definition 3.3.11. A detailed discussion exceeds the scope of this dissertation.

Note that the proof transformation discussed in this section may introduce tautological clauses $(C_1 \vee \mathtt{x} \vee \neg\mathtt{x})$. An example is provided in Figures 3.13(a) and 3.13(b). Such clauses do not contribute to the refutation and may be eliminated (as indicated in Figures 3.13(b) and 3.13(c)). The resulting reduced proof, however, may require repair by means of an algorithm such as the one presented in [BIFH+09].

In the remaining part of this section, we address the impact of proof transformations on the strength of the interpolants. Example 3.3.13 suggests that "pushing" resolution steps with pivot elements that are local to A towards the initial vertices strengthens the interpolant obtained using $\mathsf{Itp}_{\mathrm{M}}$. We generalise this idea to labelled interpolation systems.

Let L_R be a labelling function for a refutation R, $w, v \in V_R$ internal vertices and (w, v) a merge-free edge. L_R is not a valid labelling function for the proof $R[w \rightleftharpoons v]$ because $R[w \rightleftharpoons v]$ has a different clause function from R. However, R and $R[w \rightleftharpoons v]$ share the same initial vertices and labelling functions are determined by the labels of initial vertices. Accordingly, we can derive a labelling function for $R[w \rightleftharpoons v]$, denoted $L_R[w \rightleftharpoons v]$, from L_R.

The following theorem formally captures the intuition about the relation of swap transformations and interpolant strength.

Theorem 3.3.6. *Let R be a refutation of $A \wedge \overline{B}$, $v, w \in V_R$ be internal vertices with ancestors and partial interpolants as in Figure 3.15(a), and let (w, v) be a merge-free edge. For a fixed labelling function L, let* $\mathsf{c} = L(w^+, piv(w)) \sqcup L(w^-, \neg piv(w))$ *and* $\mathsf{d} = L(v^+, piv(v)) \sqcup$

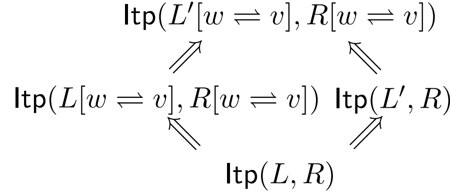


Figure 3.19: Combining labelling functions and proof transformations (Corollary 3.3.1)

$L(v^-, \neg piv(v))$. *The following claims hold:*

1. *If* $\mathsf{c} \preceq \mathsf{d}$ *and either* $\mathsf{c} \neq \mathsf{d}$ *or* $\mathsf{c} \neq \mathsf{ab}$*,* $\mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L, R)(\mathrm{s}_R)$*.*

2. *In all other cases, if* $I_2 \Rightarrow I_3$*, then* $\mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L, R)(\mathrm{s}_R)$*.*

As before, a formal proof is provided in Section B.4. The case in which the edge (w, v) is not merge-free and $l_1 \in C$ is discussed previously and does not result in a change of the interpolant. In case that $l_0 \in C_3$, however, swapping the vertices w and v may necessitate a repair of the proof, leading to the elimination of clauses and their respective partial interpolants. This global change may affect the entire proof, making a prediction of the strength of the interpolant in terms of the annotations of w, v, and their neighbouring vertices impossible.

Finally, changing labelling functions and swapping vertices are two orthogonal methods for strengthening interpolants. Corollary 3.3.1, summarised in Figure 3.19, shows that these methods can be combined.

Corollary 3.3.1. *Let* R *be a refutation of* $A \wedge \overline{B}$ *and let* L *and* L' *be labelling functions such that* $L \preceq L'$*. Let* w *and* v *be internal vertices of* R *and* (w, v) *be a merge-free edge, such that for any* L*,* $\mathsf{Itp}(L, R)(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_{R[w \rightleftharpoons v]})$*. Then, it holds that*

- $\mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L'[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_{R[w \rightleftharpoons v]})$*, and*
- $\mathsf{Itp}(L', R)(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L'[w \rightleftharpoons v], R[w \rightleftharpoons v])(\mathrm{s}_{R[w \rightleftharpoons v]})$.

The corollary follows immediately from Lemma 3.3.2, Theorem 3.3.4 and Theorem 3.3.6.

This concludes our discussion of interpolation techniques for propositional formulae. The following section establishes a connection between "bit-level" resolution proofs and "word-level" proofs in the theory of bit-vector arithmetic.

3.4 Propositional Proofs and Bit-Vector Arithmetic

Propositional interpolation systems as introduced in Section 3.3.3 provide us with formulae over propositional atoms (as specified by the grammar Table 3.2(a)). While a mapping to bit-vector formulae (see Table 3.1) based on the equality $\mathcal{E}(\mathtt{x}\&(1 \ll i)) = \mathtt{x}_i$ is straight forward, the resulting interpolants may be unwieldy. The problem is demonstrated in the following example.

Example 3.4.1. *Recall that in Example 3.3.3 we introduce propositional constraints to eliminate the assignment* $\mathtt{e}_2 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5$*, which is not ruled out by the propositional skeleton* $(\neg\mathtt{e}_1 \vee \mathtt{e}_2) \wedge \mathtt{e}_3 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5$ *in Example 3.3.1. Furthermore, in Example 3.3.5, we derive* $\mathtt{z}_0$ *(*$\mathcal{E}(\mathtt{z}\&1 = 1)$*, respectively) as a reverse interpolant for*

$$\mathtt{e}_2 \wedge \mathtt{e}_4 \wedge (\mathtt{e}_2 \Leftrightarrow \mathcal{E}(\mathtt{x}\&2 = 2)) \wedge (\mathtt{e}_4 \Leftrightarrow \mathcal{E}(\mathtt{x} = \mathtt{z} \ll 1)) \quad \textit{and} \quad \mathtt{e}_5 \wedge (\mathtt{e}_5 \Leftrightarrow \mathcal{E}(\mathtt{z}\&1 = 0))\,.$$

We demonstrate that propositional interpolants are not always as concise as in Example 3.3.5. Consider the assignment $\neg\mathtt{e}_1 \wedge \mathtt{e}_3 \wedge \mathtt{e}_4$ *of the atoms of the propositional skeleton shown above. In order to eliminate this assignment, we introduce propositional constraints for the corresponding atoms and partition the resulting formula as follows:*

$$\neg\mathtt{e}_1 \wedge (\mathtt{e}_1 \Leftrightarrow \mathcal{E}(\mathtt{x} = \mathtt{y})) \quad \textit{and} \quad \mathtt{e}_3 \wedge \mathtt{e}_4 \ \wedge \ (\mathtt{e}_3 \Leftrightarrow \mathcal{E}(\mathtt{y} = \mathtt{z} + \mathtt{z})) \wedge (\mathtt{e}_4 \Leftrightarrow \mathcal{E}(\mathtt{x} = \mathtt{z} \ll 1))$$

According to Table 3.4 and Example 3.3.2 the corresponding propositional encoding is

$$\neg\mathtt{e}_1 \ \wedge \ (\mathtt{e}_1 \vee \mathtt{o}_0 \vee \mathtt{o}_1 \vee \ldots \vee \mathtt{o}_{63}) \wedge (\mathtt{o}_0 \Leftrightarrow (\mathtt{x}_0 \wedge \neg\mathtt{y}_0)) \wedge (\mathtt{o}_{32} \Leftrightarrow (\neg\mathtt{x}_{32} \wedge \mathtt{y}_{32})) \wedge \ldots$$
$$\textit{and} \quad \mathtt{e}_3 \wedge \mathtt{e}_4 \ \wedge \ (\mathtt{e}_3 \Rightarrow \neg\mathtt{y}_0) \wedge (\mathtt{e}_4 \Rightarrow \neg\mathtt{x}_0)\,.$$

Intuitively, the propositional encoding of $\mathtt{x} \neq \mathtt{y}$ *states that the atoms of at least one of the pairs* $\mathtt{x}_i, \mathtt{y}_i$ *(where* $0 \leq i \leq 31$*) must take on opposing values. Figure 3.20 shows a small fraction of the resolution proof for the propositional encoding above, which is annotated with the partial interpolants obtained using* Itp_{HKP}*. Notably, the interpolation system introduces*

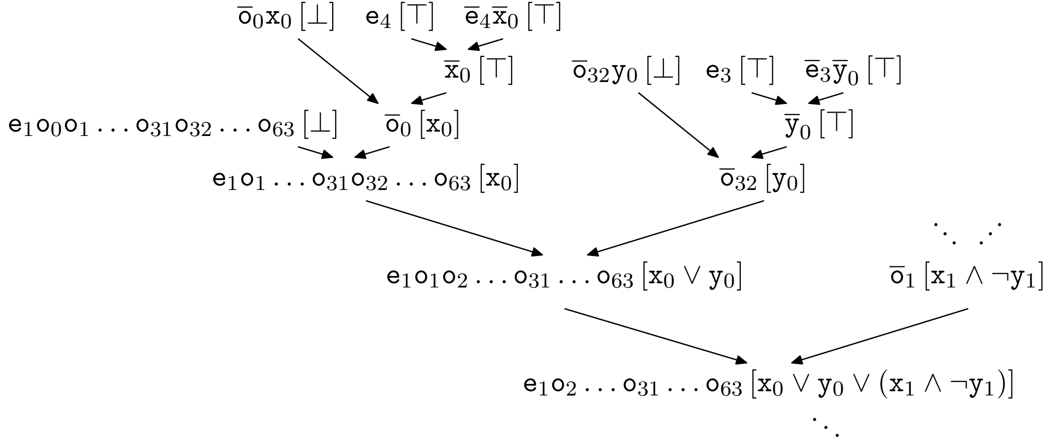


Figure 3.20: Propositional refutation of $(\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)$

a constraint for each pair x_i, y_i of bits. The resulting propositional reverse interpolant is

$$x_0 \vee y_0 \vee (x_1 \wedge \neg y_1) \vee (\neg x_1 \wedge y_1) \vee (x_2 \wedge \neg y_2) \vee \ldots \vee (x_{31} \wedge \neg y_{31}) \vee (\neg x_{31} \wedge y_{31}) \,.$$

*Note that this interpolant is not the same as $\mathcal{E}(\mathtt{x} \neq \mathtt{y})$, since it is sufficient to set either x_0 or y_0 to **true** in order to contradict $\mathcal{E}(\mathtt{y} = \mathtt{z} + \mathtt{z} \wedge \mathtt{x} = \mathtt{z} \ll 1)$.*

The propositional interpolant derived in Example 3.4.1 has several shortcomings. First of all, and most importantly, its structure does not reflect the structure of the original program statements. Therefore, the formula is incomprehensible to a human mind, which complicates the interpretation of the information provided by a safety proof. This difficulty becomes even more obvious if one considers the interpolant $\mathtt{x} \neq 0 \wedge \mathtt{y} \leq (\mathtt{x} - 1)$ in our introductory example in Section 1.3.1. Furthermore, even though the use of purely propositional formulae does not impact the correctness of the verification algorithm, the representation is significantly less compact than bit-vector formulae encoding "word-level" information such as $\mathtt{x} \neq \mathtt{y}$, which may have a negative impact on the performance and the complexity of the verification tool. Consequently, implementing and debugging the verification tool becomes a tedious and error-prone task.

On the other hand, the propositional encoding of bit-vector formulae affords us several advantages. In enables the use of off-the-shelf decision procedures (SAT solvers in particular). Modern SAT solvers deal efficiently with case-splitting (introduced by borderline cases

such as arithmetic overflow) and the propositional structure of the problem instances. For unsatisfiable instances, SAT solvers are typically able to quickly identify an *unsatisfiable core* of the formula, i.e., to discard the clauses that do not contribute to the inconsistency. Moreover, the flattening-based approach presented in Section 3.3 is complete for the quantifier-free fragment of bit-vector arithmetic specified in Table 3.1.

3.4.1 SMT-Solvers and Blocking Clauses

Contemporary interpolating decision procedures (such as [McM05, YM05, CGS10]) use *blocking clauses* to combine the advantages of theory-specific solvers and SAT solvers.[2] A blocking clause is a constraint representing the negation of a "spurious" assignment.

Example 3.4.2. *The skeleton* $(\neg\mathtt{e}_1 \vee \mathtt{e}_2) \wedge \mathtt{e}_3 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5$ *presented in Example 3.3.1 does not rule out the assignments* $\mathtt{e}_2 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5$ *and* $\neg\mathtt{e}_1 \wedge \mathtt{e}_3 \wedge \mathtt{e}_4$. *We know, however, from the Examples 3.3.3 and 3.4.1 that these assignments correspond to the unsatisfiable conjunctions* $((\mathtt{x}\&2) = 2)\wedge(\mathtt{x} = \mathtt{z} \ll 1)\wedge((\mathtt{z}\&1) = 0)$ *and* $(\mathtt{x} \neq \mathtt{y})\wedge(\mathtt{y} = \mathtt{z}{+}\mathtt{z})\wedge(\mathtt{x} = \mathtt{z} \ll 1)$. *Accordingly, the clauses* $(\neg\mathtt{e}_2 \vee \neg\mathtt{e}_4 \vee \neg\mathtt{e}_5)$ *and* $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ *(which are the negations of the above mentioned conjunctions) are a consequence of*

$$(\neg\mathtt{e}_1 \vee \mathtt{e}_2) \wedge \mathtt{e}_3 \wedge \mathtt{e}_4 \wedge \mathtt{e}_5 \;\wedge\; (\mathtt{e}_1 \Leftrightarrow (\mathtt{x} = \mathtt{y})) \wedge (\mathtt{e}_2 \Leftrightarrow ((\mathtt{x}\&2) = 2))\wedge$$
$$(\mathtt{e}_3 \Leftrightarrow (\mathtt{y} = \mathtt{z} + \mathtt{z})) \wedge (\mathtt{e}_4 \Leftrightarrow (\mathtt{x} = \mathtt{z} \ll 1)) \wedge (\mathtt{e}_5 \Leftrightarrow (\mathtt{z}\&1 = 0)) \quad (3.7)$$

and can therefore be added to the skeleton in order to eliminate (or "block") the undesired assignments. A resolution proof using the blocking clauses $(\neg\mathtt{e}_2 \vee \neg\mathtt{e}_4 \vee \neg\mathtt{e}_5)$ *and* $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ *is shown in Figure 3.21.*

In Example 3.4.2, the encoding (3.7) associates each blocking clause with an unsatisfiable conjunction of literals in the theory of bit-vectors. In general, given a formula F, the bijective mapping between propositional literals $\mathrm{Atoms}(\mathsf{sk}(F))$ and theory literals $\mathrm{Atoms}^{\mathcal{T}}(F)$ induced by the propositional skeleton (Definition 3.3.1) enables us to map blocking clauses

[2] The survey [NOT06] and the textbook [Har09] attribute the first approach basing other decision procedures around SAT to Armando, Castellini, and Giunchiglia [ACG00].

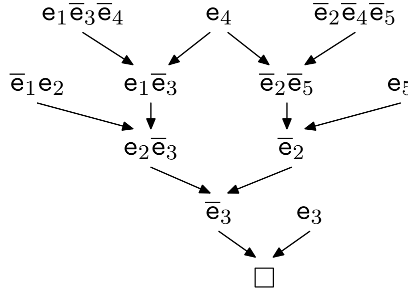


Figure 3.21: A resolution proof with blocking clauses

to conjunctions of theory literals. Assuming the existence of a complete interpolating decision procedure for the conjunctive fragment of the theory $\mathcal{T}$, [McM05, YM05] provides us with a technique to derive partial interpolants (cf. Section 3.2.3, Definition 3.2.4) for blocking clauses. We adapt this concept to accommodate the partial interpolants introduced in Definition 3.3.12.

Lemma 3.4.1 (Partial Interpolant for Blocking Clause)**.** *Let $A \wedge \overline{B}$ be an unsatisfiable bit-vector formula (as specified by the grammar in Table 3.1) and let $\mathcal{T}$ be the underlying theory of bit-vector arithmetic. Moreover, let*

$$\mathsf{sk}(A) \wedge \bigwedge_{\mathsf{e}_i \in \text{Atoms}(\mathsf{sk}(A))} (\mathsf{e}_i \Leftrightarrow \alpha_i) \quad \textit{and} \quad \mathsf{sk}(\overline{B}) \wedge \bigwedge_{\mathsf{e}_j \in \text{Atoms}(\mathsf{sk}(\overline{B}))} (\mathsf{e}_j \Leftrightarrow \alpha_j) \tag{3.8}$$

be encodings equi-satisfiable with A and $\overline{B}$, respectively. Given a blocking clause

$$C \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} l_i \quad \textit{where} \quad l_i \in \{\mathsf{e}, \neg\mathsf{e} \mid \mathsf{e} \in \text{Atoms}(\mathsf{sk}(A \wedge \overline{B}))\}$$

for the propositional skeleton $\mathsf{sk}(A \wedge \overline{B})$, let

$$C^{\mathcal{T}} \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} l_i^{\mathcal{T}} \quad \textit{where} \quad l_i^{\mathcal{T}} \in \{\alpha, \neg\alpha \mid \alpha \in \text{Atoms}^{\mathcal{T}}(A \wedge \overline{B})\}$$

be the corresponding disjunction of theory literals. Analogously, let $C^{\mathcal{T}}\restriction_{\mathsf{c}}$ denote the conjunction of theory literals corresponding to the clause $C\restriction_{\mathsf{c}}$. Finally, let I be a reverse interpolant for the pair of bit-vector formulae $(\neg(C^{\mathcal{T}}\restriction_{\mathsf{a}}), \neg(C^{\mathcal{T}}\restriction_{\mathsf{b}}))$. Then I is a partial interpolant for C and the encodings (3.8) of A and $\overline{B}$.

Proof. We need to show that I satisfies the conditions listed in Definition 3.3.12.

1. $\left(\mathsf{sk}(A) \wedge \bigwedge_{\mathsf{e}_i \in \mathrm{Atoms}(\mathsf{sk}(A))}(\mathsf{e}_i \Leftrightarrow \alpha_i)\right) \wedge \neg(C\lceil_{\mathsf{a}}) \models_{\mathcal{T}} I$. This follows immediately from $\left(\bigwedge_{\mathsf{e}_i \in \mathrm{Atoms}(\mathsf{sk}(A))}(\mathsf{e}_i \Leftrightarrow \alpha_i)) \wedge \neg(C\lceil_{\mathsf{a}})\right) \Rightarrow \neg(C^{\mathcal{T}}\lceil_{\mathsf{a}})$.

2. $\left(\mathsf{sk}(\overline{B}) \wedge \bigwedge_{\mathsf{e}_j \in \mathrm{Atoms}(\mathsf{sk}(\overline{B}))}(\mathsf{e}_j \Leftrightarrow \alpha_j)\right) \wedge \neg(C\lceil_{\mathsf{b}}) \models_{\mathcal{T}} \neg I$. This follows immediately from $\left(\bigwedge_{\mathsf{e}_j \in \mathrm{Atoms}(\mathsf{sk}(\overline{B}))}(\mathsf{e}_j \Leftrightarrow \alpha_j)) \wedge \neg(C\lceil_{\mathsf{b}})\right) \Rightarrow \neg(C^{\mathcal{T}}\lceil_{\mathsf{b}})$.

3. The symbols in $\neg(C^{\mathcal{T}}\lceil_{\mathsf{a}})$ and $\neg(C^{\mathcal{T}}\lceil_{\mathsf{b}})$ are subsets of the symbols in A and $\overline{B}$. Accordingly, the symbols occurring in I are shared symbols of A and $\overline{B}$.

□

Note that the labels assigned to the literals of blocking clauses need to be consistent with the selected labelling scheme (see, for instance, the labelling functions in Lemma 3.3.1). For example, all shared skeleton literals must be labelled b if the partial interpolants for blocking clauses are to be combined with the interpolation system $\mathsf{Itp}_{\mathrm{M}}$.

It remains to be shown how we can obtain blocking clauses and how they can be integrated into an resolution-based solver for propositional logic. The simplest approach to combine a theory-specific decision procedure for the theory $\mathcal{T}$ with propositional reasoning is to treat the SAT checker as an oracle which generates satisfying assignments. The following algorithm summarises the principles of a basic *Satisfiability Modulo Theory* (SMT) approach.

① Given a bit-vector formula F, generate the propositional skeleton (Definition 3.3.1) $\mathsf{sk}(F)$ by replacing all theory-specific atoms $\mathrm{Atoms}^{\mathcal{T}}(F)$ with propositional atoms. Furthermore, introduce a constraint Γ with the initial value true.

② If the formula $\mathsf{sk}(F) \wedge \Gamma$ is unsatisfiable, then F is unsatisfiable. Otherwise, let $\bigwedge_{i=1}^{|\mathrm{Atoms}(F)|} l_i$ represent an assignment to the atoms $\mathrm{Atoms}(\mathsf{sk}(F))$ satisfying $\mathsf{sk}(F)$.

③ Apply the mapping between $\mathrm{Atoms}(\mathsf{sk}(F))$ and $\mathrm{Atoms}^{\mathcal{T}}(F)$ in order to construct a conjunction of theory-specific literals which corresponds to $\bigwedge_{i=1}^{|\mathrm{Atoms}(F)|} l_i$. Use the theory-specific solver to check whether this conjunction is satisfiable in the theory $\mathcal{T}$.

If it is satisfiable, then so is F. Otherwise, add the clause $\neg \bigwedge_{i=1}^{|\text{Atoms}(F)|} l_i$ to Γ and proceed with step ②.

This technique shifts the burden of case-splitting from the theory specific solver to the SAT solver. In the worst case, however, the implementation outlined above naïvely enumerates all satisfying assignments to $\mathsf{sk}(F)$, effectively generating a representation of the formula in disjunctive normal form (DNF). In order to avoid an exponential blowup of the encoding, theory solvers are integrated into the SAT solver in a more sophisticated way. For instance, the solver may add additional clauses representing intermediate proof steps such as

$$(\neg \mathsf{e}_3 \vee \mathsf{e}_6) \quad \text{for} \quad (\mathtt{y} = \mathtt{z} + \mathtt{z}) \vdash_{\mathcal{T}} (\mathtt{y} = \mathtt{z} \ll 1) \quad \text{and}$$

$$(\mathsf{e}_1 \vee \neg \mathsf{e}_6 \vee \neg \mathsf{e}_4) \quad \text{for} \quad (\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} \ll 1) \vdash_{\mathcal{T}} (\mathtt{x} \neq \mathtt{z} \ll 1)$$

in order to aid the search of the SAT solver [FJOS03]. We refer the reader to [NOT06] for a detailed discussion of SMT solvers. A tighter integration of SAT checkers and theory solvers (known as *online* approach), however, requires modifications to the theory solver.

3.4.2 Lifting Propositional Resolution Proofs to the Word-Level

In this section, we discuss an *ex post facto* approach which relies entirely on off-the-shelf interpolating decision procedures. Essentially, we take advantage of the SAT solver's ability to efficiently extract a small, unsatisfiable core from a propositional encoding. Our idea is to derive blocking clauses from the corresponding resolution proof. Given these clauses, we compute the respective partial interpolants using a theory specific solver.

The core component of our approach is an algorithm that *lifts* a propositional resolution proof R up to a proof that uses bit-vector logic.[3] The structure of R remains unchanged (up to a final minimisation step). We define a lifting function λ which annotates each vertex in the resolution refutation R with a formula consisting of skeleton literals and theory literals. Each of the resulting annotations has the structure $\Theta \vee \Gamma$, where Θ is a disjunction of skeleton literals and Γ represents a Boolean combination of theory atoms $\alpha_1, \ldots, \alpha_n$.

In the following definition, we use $\mathsf{Tseitin}(F)$ to denote the formula in CNF obtained

[3]We present a similar proof-lifting approach for a very limited subset of bit-vector logic in [KW07].

from a propositional formula F using Tseitin's encoding (see Table 3.3). As explained in Section 3.3.1, $\mathsf{Tseitin}(F)$ and F are equi-satisfiable.

Definition 3.4.1 (Lifting Function)**.** *Let F be an unsatisfiable bit-vector formula and let $R = (V_R, E_R, piv_R, \ell_R, \mathrm{s}_R)$ be a refutation of its propositional encoding*

$$\mathsf{Tseitin}(\mathsf{sk}(F)) \;\wedge \bigwedge_{\mathsf{e}_i \in \mathrm{Atoms}(\mathsf{sk}(F))} \mathsf{Tseitin}(\mathsf{e}_i \Rightarrow \mathcal{E}(\alpha_i)) \wedge \mathsf{Tseitin}(\mathcal{E}(\alpha_i) \Rightarrow \mathsf{e}_i) \tag{3.9}$$

(where $\alpha_i \in \mathrm{Atoms}^{\mathcal{T}}(F)$). The lifting function $\lambda : V_R \rightarrow \mathcal{L}$ for R is defined as follows:

1. *For all initial vertices $v \in V_R$*

$$\lambda(v) \stackrel{\text{def}}{=} \begin{cases} \neg\mathsf{e}_i \vee \alpha_i & \text{if } \ell(v) \in \mathsf{Tseitin}(\mathsf{e}_i \Rightarrow \mathcal{E}(\alpha_i)) \\ \mathsf{e}_i \vee \neg\alpha_i & \text{if } \ell(v) \in \mathsf{Tseitin}(\mathcal{E}(\alpha_i) \Rightarrow \mathsf{e}_i) \\ \ell(v) & \text{if } \ell(v) \in \mathsf{Tseitin}(\mathsf{sk}(F)) \end{cases}$$

2. *For all inner vertices $v \in V_R$ with $(v^-, v) \in E_R$ and $(v^+, v) \in E_R$ and $\lambda(v^-) = \Theta^- \vee \Gamma^-$ and $\lambda(v^+) = \Theta^+ \vee \Gamma^+$*

$$\lambda(v) \stackrel{\text{def}}{=} \begin{cases} (\Theta^- \vee \Theta^+) \vee (\Gamma^- \wedge \Gamma^+) & \text{if } piv(v) \notin \mathrm{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F))) \\ \mathrm{Res}(\Theta^-, \Theta^+, piv(v)) \vee (\Gamma^- \vee \Gamma^+) & \text{if } piv(v) \in \mathrm{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F))) \end{cases}$$

The lifting function replaces the propositional formulae at the leaves which are derived from the constraints $\mathsf{e} \Rightarrow \mathcal{E}(\alpha)$ and $\neg\mathsf{e} \Rightarrow \neg\mathcal{E}(\alpha)$ (cf. Equation 3.4 in Section 3.3.1) with the original word-level constraints. Then, it essentially creates a new proof which replicates the resolution steps on the skeleton literals and simulates the remaining inference steps by conjoining the respective word-level literals.

We show that the "lifted" proof is indeed valid.

Lemma 3.4.2. *Let R be a resolution proof as specified in Definition 3.4.1. For each inner vertex $v \in V_R$ with $(v^-, v) \in E_R$ and $(v^+, v) \in E_R$ it holds that $\lambda(v^-) \wedge \lambda(v^+) \Rightarrow \lambda(v)$.*

Proof. Let $v \in V_R$ be an inner vertex such that $\lambda(v^-) = \Theta^- \vee \Gamma^-$ and $\lambda(v^+) = \Theta^+ \vee \Gamma^+$. Thus, $\lambda(v-) \wedge \lambda(v^+) \equiv (\Theta^- \vee \Gamma^-) \wedge (\Theta^+ \vee \Gamma^+)$. By means of the distribution rule of the propositional calculus we obtain

$$(\Theta^- \wedge \Theta^+) \vee (\Theta^- \wedge \Gamma^+) \vee (\Theta^+ \wedge \Gamma^-) \vee (\Gamma^- \wedge \Gamma^+) \tag{3.10}$$

We need to distinguish two cases.

1. $piv(v) \notin \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$. The conjuncts in (3.10) imply either Θ^-, Θ^+ or $(\Gamma^- \wedge \Gamma^+)$.

2. $piv(v) \in \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$. The conjuncts in (3.10) imply either Γ^-, Γ^+ or $(\Theta^- \wedge \Theta^+)$. The latter term in turn implies $\text{Res}(\Theta^-, \Theta^+, piv(v))$.

□

The following lemma establishes a semantic connection between the labels $\ell(v)$ and $\lambda(v)$.

Lemma 3.4.3. *Let R be a resolution proof as described in Definition 3.4.1. For any vertex $v \in V_R$ with $\lambda(v) = \Theta \vee \Gamma$, it holds that if $\mathcal{E}(\Gamma)$ is satisfiable, then so is $\ell(v) \setminus \Theta$.*

Proof. We use induction over the structure of R to prove the correctness of Lemma 3.4.3.

Base case. Let $v \in V_R$ be an initial vertex and let $\lambda(v) = \Theta \vee \Gamma$. In order to show that the satisfiability of $\mathcal{E}(\Gamma)$ entails that there is a satisfying assignment to $\ell(v) \setminus \Theta$, we need to distinguish three cases. If $\ell(v) \in \mathsf{Tseitin}(\mathsf{sk}(F))$ then $\Theta = \ell(v)$ and $\Gamma = \mathsf{false}$ and the claim holds trivially. Otherwise, if $\ell(v) \in \mathsf{Tseitin}(\mathsf{e}_i \Rightarrow \mathcal{E}(\alpha_i))$, then $\Theta = \neg\mathsf{e}_i$ and $\Gamma = \alpha_i$. W.l.o.g. we assume that $\mathsf{Tseitin}(\mathsf{e}_i \Rightarrow \mathcal{E}(\alpha_i))$ yields $(\neg\mathsf{e}_i \vee \mathsf{o}_i) \wedge \mathsf{Tseitin}(\mathsf{o}_i \Leftrightarrow \mathcal{E}(\alpha_i))$. If $\ell(v) = (\neg\mathsf{e}_i \vee \mathsf{o}_i)$ then $\ell(v) \setminus \Theta = \mathsf{o}_i$, and $\{\mathsf{o}_i \mapsto \mathsf{true}\}$ is a satisfying assignment. Moreover, since $\mathsf{o}_i \Leftrightarrow \mathcal{E}(\alpha_i)$ can always be satisfied by choosing an appropriate value for o_i, any conjunct of the equi-satisfiable Tseitin encoding must be satisfiable, too. A similar argument holds for the remaining case $\ell(v) \in \mathsf{Tseitin}(\mathcal{E}(\alpha_i) \Rightarrow \mathsf{e}_i)$.

Induction hypothesis. Let $\lambda(v^-) = \Theta^- \vee \Gamma^-$ and $\lambda(v^+) = \Theta^+ \vee \Gamma^+$. If $\mathcal{E}(\Gamma^-)$ is satisfiable, then so is $\ell(v^-) \setminus \Theta^-$. The same holds for $\mathcal{E}(\Gamma^+)$ and $\ell(v^+) \setminus \Theta^+$.

Induction step. Let $v \in V_R$ be an internal vertex where $(v^-, v) \in E_R$ and $(v^+, v) \in E_R$ and $\lambda(v^-) = \Theta^- \vee \Gamma^-$ and $\lambda(v^+) = \Theta^+ \vee \Gamma^+$.

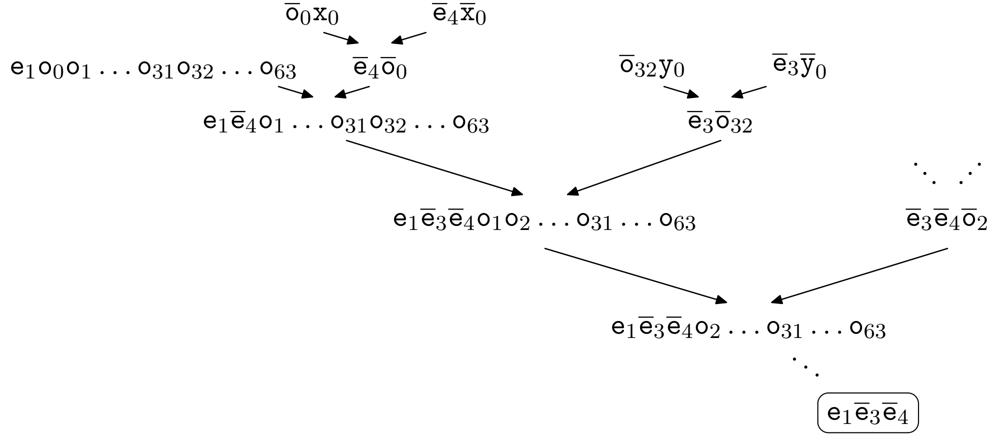
1. If $piv(v) \notin \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$, then $\lambda(v) = (\Theta^- \vee \Theta^+) \vee (\Gamma^- \wedge \Gamma^+)$. Since R is a valid resolution proof, $\ell(v) = \text{Res}(\ell(v^-), \ell(v^+), piv(v))$ holds. Suppose that $\mathcal{E}(\Gamma^- \wedge \Gamma^+)$ is satisfiable. Then the bit-vector formula $\Gamma^- \wedge \Gamma^+$ is satisfiable, and so are the formulae $\mathcal{E}(\Gamma^-)$ and $\mathcal{E}(\Gamma^+)$. By the induction hypothesis, there exist satisfying assignments for $\ell(v^-) \setminus \Theta^-$ as well as for $\ell(v^+) \setminus \Theta^+$. We show that this premise is sufficient to construct a satisfying assignment for $\ell(v)$: W.l.o.g. let $\mathcal{M}^-$ and $\mathcal{M}^+$ be models such that $\mathcal{M}^- \models l^-$ and $\mathcal{M}^+ \models l^+$, where $l^- \in (\ell(v^-) \setminus \Theta^-)$ and $l^+ \in (\ell(v^+) \setminus \Theta^+)$. If $l^- \in \ell(v) \setminus (\Theta^- \vee \Theta^+)$, then $\mathcal{M}^-$ can be extended to a model that satisfies $\ell(v) \setminus (\Theta^- \vee \Theta^+)$. Otherwise, $\text{Atoms}(l^-) = \{piv(v)\}$. Then $l^+ \neq \neg l^-$, since $l^- \mapsto \mathsf{true}$. Accordingly, $l^+ \in \ell(v) \setminus (\Theta^- \vee \Theta^+)$ and $\mathcal{M}^+$ can be extended to a model satisfying $\ell(v) \setminus (\Theta^- \vee \Theta^+)$.

2. If $piv(v) \in \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$, then $\lambda(v) = \text{Res}(\Theta^-, \Theta^+, piv(v)) \vee (\Gamma^- \vee \Gamma^+)$. Suppose that $\mathcal{E}(\Gamma^- \vee \Gamma^+)$ is satisfiable. Then the bit-vector formula $\Gamma^- \vee \Gamma^+$ is satisfiable, and so is at least one of the formulae $\mathcal{E}(\Gamma^-)$ and $\mathcal{E}(\Gamma^+)$. By the induction hypothesis, there exists an assignment $\mathcal{M}$ satisfying either $\ell(v^-) \setminus \Theta^-$ or $\ell(v^+) \setminus \Theta^+$ (or both). Since $piv(v) \in \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$, it holds that $(\ell(v^-) \vee \ell(v^+)) \setminus (\Theta^- \vee \Theta^+) \subseteq \ell(v)$, and therefore $\mathcal{M}$ can be extended to a model satisfying $\ell(v) \setminus (\Theta^- \vee \Theta^+)$.

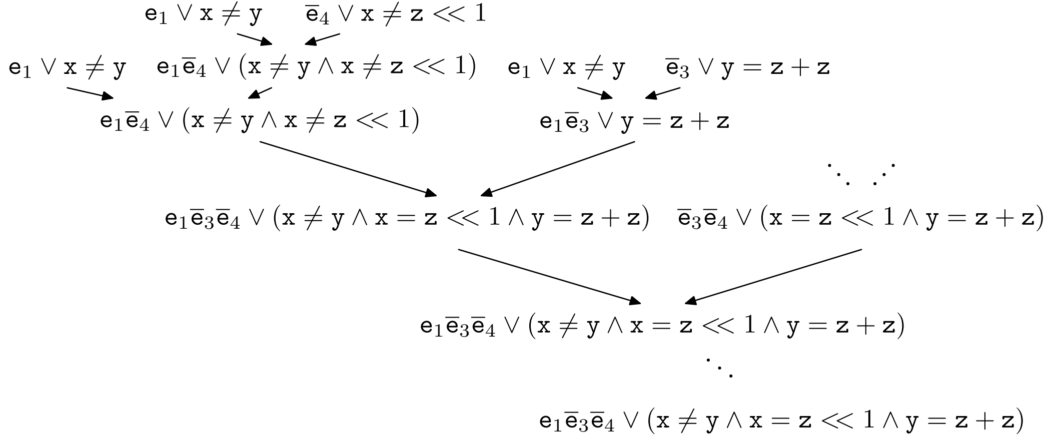
□

Corollary 3.4.1. *Let R be a refutation proof as specified in Definition 3.4.1 and let $v \in V_R$ be a vertex of R. If $\lambda(v) = \Theta \vee \Gamma$ and $\ell(v) = \Theta$, then Γ is unsatisfiable.*

Corollary 3.4.1 provides us with a mechanism to derive blocking clauses from propositional resolution proofs. Let F be a bit-vector arithmetic formula and let R be a resolution refutation of the corresponding propositional encoding resembling (3.9) in Definition 3.4.1. Given a vertex $v \in V_R$ such that $\lambda(v) = \Theta \vee \Gamma$ and $\ell(v) = \Theta$, we know that Γ is a Boolean combination of theory atoms in $\text{Atoms}^{\mathcal{T}}(F)$ which is unsatisfiable in the theory



(a) A resolution proof for the propositional encoding of the formula in Example 3.3.1



(b) Result of the lifting function λ for the proof in Figure 3.22(a)

Figure 3.22: Deriving blocking clauses from propositional resolution proofs

of bit-vectors. The bijective mapping induced by the propositional encoding enables us to construct a corresponding formula over the atoms $\text{Atoms}(\mathsf{sk}(F))$.

Example 3.4.3. *Let R be a resolution refutation of the propositional encoding of the formula*

$$(\neg(\mathtt{x} = \mathtt{y}) \vee ((\mathtt{x}\&2) = 2)) \;\wedge\; (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$$

presented in Example 3.3.1. The propositional encoding is described in detail in Table 3.4 in Section 3.3. Figure 3.22(a) shows a branch of such a proof R. Figure 3.22(b) shows the "word-level proof" which we obtain by applying λ to R. Consider the vertex v labelled $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ *(emphasised in Figure 3.22(a)). The corresponding label in Figure 3.22(b)*

is

$$\lambda(v) = (\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4) \vee (\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)\,.$$

Accordingly, $(\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)$ *is unsatisfiable in the theory of bit-vectors. The corresponding propositional formula is* $\neg\mathtt{e}_1 \wedge \mathtt{e}_3 \wedge \mathtt{e}_4$*. Consequently, the negation* $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ *of this formula is a blocking clause. The corresponding partial interpolant (as defined in Lemma 3.4.1) with respect to the partitioning introduced in Example 3.4.1 is* $\mathtt{x} \neq \mathtt{y}$*.*

Suppose we can derive the blocking clause $(\neg\mathtt{e}_2 \vee \neg\mathtt{e}_4 \vee \neg\mathtt{e}_5)$ *(cf. Example 3.4.2) in a similar manner. If the clauses* Θ *derived by the respective sub-proofs (as in Figure 3.22(a)) match the blocking clauses, we can replace the sub-proofs by the blocking clauses in order to obtain a resolution proof over the skeleton literals such as the one illustrated in Figure 3.21.*

The resolution proof in Figure 3.22(a) of Example 3.4.3 illustrates the special case in which the resolution steps involving the skeleton clauses $C \in \mathsf{Tseitin}(\mathsf{sk}(F))$ are "postponed" until the point when all literals $l \notin \text{Atoms}(\mathsf{Tseitin}(\mathsf{sk}(F)))$ have been eliminated. Let v be the vertex in Figure 3.22(a) labelled $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ and let $\lambda(v) = \Theta \vee \Gamma$. We observe that Γ is a conjunction of theory literals and that Θ is a blocking clause (as assumed in Example 3.4.3). In general, this is not necessarily the case:

1. Consider the resolution proof in Figure 3.20 and let v be the node for which $\ell(v) = \mathtt{e}_1$ holds, i.e., all atoms $\mathtt{o}_i$ have been eliminated. Then the labelling function λ yields $\lambda(v) = \mathtt{e}_1 \vee (\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)$. The corresponding blocking clause is $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$.

2. Figure 3.23 shows a fragment of a resolution proof for which the labelling function λ introduces a disjunction in Γ.

In the first case, we cannot simply replace the sub-proof deriving $\mathtt{e}_1$ by the blocking clause $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ anymore to obtain a resolution refutation over skeleton literals (as suggested in Example 3.4.3). This is only possible if the proof adheres to certain structural restrictions. In particular, all skeleton literals need to be eliminated *after* the literals used to encode the theory atoms (i.e., the literals introduced via the encoding $\mathcal{E}(\alpha_i)$). Intuitively,

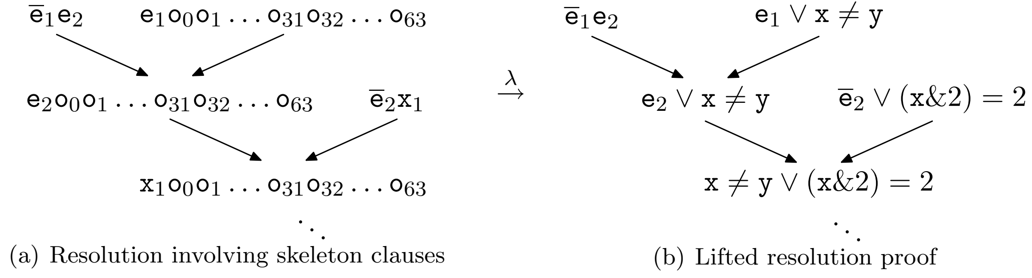


Figure 3.23: Lifting of a resolution proof introducing a disjunction in Γ

this requirement resembles the locality condition in Definition 3.2.2. In order to obtain a valid resolution proof that enables us to derive an interpolant using blocking clauses, we have two options:

- The proof transformation techniques presented in Section 3.3.5 enable us to restructure the propositional resolution proof at the cost of a potential exponential increase of the size of the proof (in the presence of merge literals, cf. Figure 3.17).
- Given the blocking clauses and the propositional skeleton, we can enquire the SAT solver to provide a corresponding resolution proof. While this approach corresponds to a partial reconstruction of the original resolution proof, only the skeleton literals relevant to the contradiction are considered. The SAT solver is confronted with a problem of reduced complexity instead of the original problem.

In the case in which Γ is not a conjunction, it is not possible to derive an interpolant from Γ using an interpolating solver supporting only a conjunctive fragment of $\mathcal{L}$ (as presumed in Section 3.4.1). Since, however, we know that Γ is unsatisfiable in the theory of bit-vector arithmetic, we can fall back on the basic SMT-solving algorithm presented in Section 3.4.1. In the worst case, this corresponds to a conversion of Γ into an equivalent formula Γ_{DNF} in disjunctive normal form. Accordingly, we may obtain more than one blocking clause; one for each conjunct of the formula Γ_{DNF}. Since the symbolic simulation techniques presented in Section 2.5.3 predominantly yield conjunctive formulae and since the SAT solver eliminates case splits not contributing to the contradiction upfront, this number is typically small in the context of our application.

Finally, we demonstrate how "bit-level" and "word-level" reasoning can be combined.

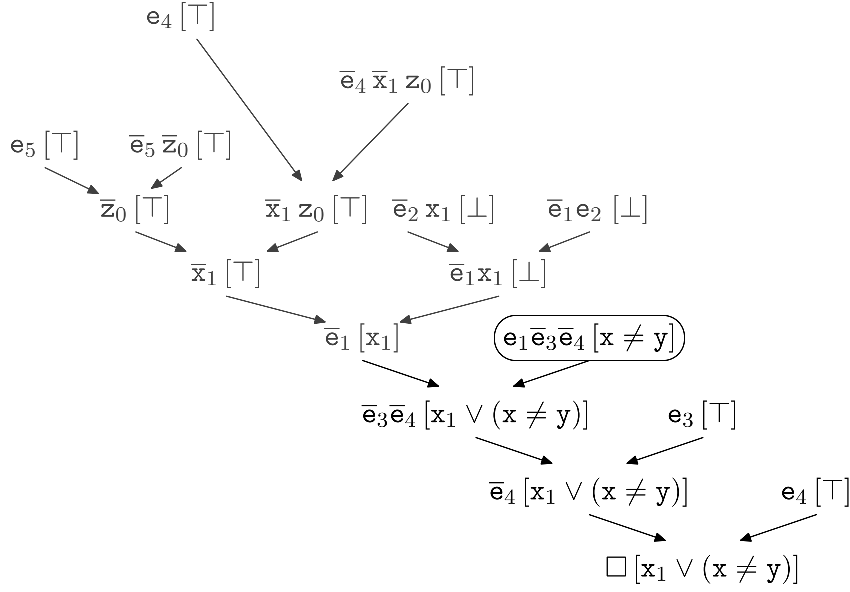


Figure 3.24: A propositional proof with a blocking clause

Given a bit-vector formula F, the propositional encoding discussed in Section 3.3.1 generates an equi-satisfiable propositional formula in which each symbol $\mathtt{x}$ of F is represented by a set of propositional symbols $\mathtt{x}_0, \ldots, \mathtt{x}_{n-1}$. The transformation preserves the bit-vector semantics in the sense that there is a direct correspondence between $\mathtt{x}_i$ and $\mathtt{x}\&(1 \ll i)$. Accordingly, for each model satisfying the propositional encoding $\mathcal{E}(F)$ the corresponding model in the domain of bit-vectors satisfies the original formula F. Intuitively, $\mathcal{E}(F)$ "implies" F (though formally the implication for formulae taking values in two different domains is not defined). If we allow this liberal notion of implication, then an interpolant for the pair of bit-vector formulae (A, B) is also an interpolant for $(\mathcal{E}(A), \mathcal{E}(B))$. This observation enables us to "mix" bit-level and word-level interpolants.

Example 3.4.4. *Figure 3.24 shows a resolution refutation of the propositional encoding of the formula (3.2). We annotate the proof with partial interpolants for the partitioning* $A \equiv (\neg(\mathtt{x} = \mathtt{y}) \vee ((\mathtt{x}\&2) = 2))$ *and* $\overline{B} \equiv (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$*. By abuse of notation, we annotate the blocking clause* $(\mathtt{e}_1 \vee \neg\mathtt{e}_3 \vee \neg\mathtt{e}_4)$ *with the partial interpolant* $\mathtt{x} \neq \mathtt{y}$ *(derived in Figure 3.22(b)). The resulting "mixed" interpolant* $\mathtt{x}_1 \vee \mathtt{x} \neq \mathtt{y}$ *happens to be "equivalent" to* A*.*

The following section is concerned with the construction of interpolants for the conjunc-

tive fragment of bit-vector arithmetic, a technique which we presumed to be a given in the current section.

3.5 Interpolation for Word-Level Bit-Vector Arithmetic

The interpolation technique presented in Section 3.4 relies on an interpolating decision procedure for a conjunctive fragment of bit-vector arithmetic. Example 3.1.1 on page 76 demonstrates that such a decision procedure is only sound if its interpretation of arithmetic operations is consistent with the finite domain semantics of bit-vector arithmetic. Many existing verification tools (such as Blast [HJMS02, HJMM04], Slam [BCLR04], and Impact [McM06]), however, approximate the semantics of bit-vector operations using decision procedures for linear arithmetic or difference logic over an unbounded domain (e.g., [McM05, RSS07, CGS10]). While interpolation techniques for modular arithmetic (such as the one presented in [JCG09]) provide the required accuracy, they lack support for operations such as bit-wise disjunction or conjunction.

The interpolation technique discussed in this section avoids the discrepancy between the bit-vector interpretation underlying most programming languages and the domains $\mathbb{R}$ or $\mathbb{Z}$ used by many interpolating decision procedures. We achieve this goal, however, at the cost of completeness. We rely on a limited set of inference rules to reduce bit-vector arithmetic formulae to equality logic with uninterpreted functions (EUF) and transitive relations. Interpolation techniques for EUF are presented in [McM05, FGG+09]. The algorithms presented there extract interpolants from local refutations (as introduced in Definition 3.2.2 in Section 3.2). In Appendix A we present a graph-based decision procedure for bit-vector arithmetic which provides us with local refutations. The following section presents the inference rules forming the basic building blocks of the local refutations. Section 3.5.2 describes an algorithm to derive interpolants from such proofs.

3.5.1 Inference Rules for Bit-Vector Arithmetic

The premises of the inference rules presented in this section are theory literals, i.e., theory atoms or their negations. The conjunctive fragment of the language $\mathcal{L}$ is defined in Table 3.5.

Table 3.5: The conjunctive fragment of bit-vector arithmetic (see Table 3.1)

$$\begin{array}{rcl} \textit{cformula} & ::= & \textit{cformula} \wedge \textit{cformula} \mid \textit{literal} \\ \textit{literal} & ::= & \textit{atom} \mid \neg\textit{atom} \end{array}$$

The non-terminal *atom* is defined in Table 3.1. As mentioned in Section 3.1, the set of atoms over the relations $\rhd \in \{=, \geq, >, \neq\}$ is closed under negation. The literal $\neg(\mathtt{x} > \mathtt{y})$, for instance, can be rewritten to $\mathtt{y} \geq \mathtt{x}$. Therefore, each formula can be reduced to a normal form, a conjunction of atoms. For convenience, we also use the relations $\leq$ and $<$ in our presentation, which can be easily replaced by $\geq$ and $>$ by swapping the operands. We refer to $>$ as inequality and to $\neq$ as dis-equality in order to avoid the confusion of these two relations.

Our (incomplete) decision procedure is described in detail in Appendix A. It aims at generating a local refutation of a given conjunction of theory atoms (partitioned into two formulae A and $\overline{B}$) using the following techniques:

- It applies axiom instantiation and inference rules such as the ones presented in Tables 3.6(a) and 3.6(b) to terms containing bit-vector operations whenever possible. For instance, the occurrence of a term $\mathtt{x} + 1$ gives rise to the dis-equality $\mathtt{x} \neq \mathtt{x} + 1$. Note that in the theory of bit-vectors it would be unsound to conclude $\mathtt{x} \leq \mathtt{x} + 1$. Our axiom instantiation and inference rules guarantee that the uninterpreted symbols occurring in the premises are not "mixed" in the conclusion. The sub-terms of the atom $\mathtt{x} \geq \mathtt{z}$, which derives from the equality $\mathtt{x} = \mathtt{y} \,|\, \mathtt{z}$, for instance, have the same locality (with respect to the partitioning of the formula) as the terms of the premise.

- It uses constant propagation to derive sub-terms that are free of uninterpreted symbols. Constant terms (such as $(4 \ll 1) + 2$) are then evaluated and the decision procedure adds a corresponding tautology ($(4 \ll 1) + 2 = 10$) to the original formula. Since constants are interpreted symbols, the tautology can be added to either partition of the original formula (cf. Theorem 3.2.1).

- The decision procedure uses the rules in Table 3.7 to derive contradictions. This set of rules is complete for equality logic with uninterpreted functions. The table

Table 3.6: Axioms and inference rules for reducing $\mathcal{L}$ to EUF with $\geq$ and $>$

(a) Examples for axioms for m-bit variables

$(t_2 + c) \neq t_2$	if $c \neq 0 \bmod 2^m$	$(t \ll c) = (t + t)$	if $c = 1$
$(t_2 + c) = t_2$	if $c = 0 \bmod 2^m$	$(t \ll c) = (2^c \cdot t)$	if $1 < c < m$

(b) Examples of inference rules for bit-vector operations over the unsigned integers

$$\frac{t_1 = t_2 \,\&\, t_3}{t_1 \leq t_2} \qquad \frac{t_1 = t_2 \,\&\, t_3}{t_1 \leq t_3} \qquad \frac{t_1 = t_2 \,|\, t_3}{t_1 \geq t_2} \qquad \frac{t_1 = t_2 \,|\, t_3}{t_1 \geq t_3}$$

$$\frac{t_1 + t_2 = t_1}{t_2 = 0} \qquad \frac{0 \geq t_1}{t_1 = 0} \qquad \frac{1 > t_1}{t_1 = 0} \qquad \frac{t_1 \neq 0 \qquad 0 \geq t_1}{\textsf{false}}$$

$$\frac{t_1 \neq 0 \qquad t_2 \leq (t_1 - 1)}{t_1 \neq t_2} \qquad \frac{t_1 > t_3 \qquad t_3 \neq 0 \qquad t_2 \leq (t_1 - t_3)}{t_1 \neq t_2}$$

accompanying the Trans rule indicates how two transitive relations are combined. For instance, the atoms $\mathtt{x} > \mathtt{y}$ and $\mathtt{y} \geq \mathtt{z}$ entail $\mathtt{x} > \mathtt{z}$. Each chain of transitive relations can be partitioned into sub-chains deriving from either partition A or partition B (cf. [McM05]), enabling the construction of local derivations.

- Bit-vector operations which the procedure fails to rewrite are replaced with uninterpreted function symbols. Moreover, uninterpreted functions are used in our implementation to model array accesses (which are common in software programs but ignored in Table 3.1). The congruence closure rule requires special attention, since it may introduce an equality $f(t_1) = f(t_2)$ where either $f(t_1)$ or $f(t_2)$ are terms of "mixed" locality. It is, however, still possible to transform a proof which uses the Cong rule into a local proof [YM05, FGG$^+$09]. We provide more details in Appendix A.

The following example provides a local proof for the bit-vector formula we encountered in the Examples 3.4.2 and 3.4.3.

Example 3.5.1. *Let $A \wedge \overline{B}$ the unsatisfiable conjunction of the pair of bit-vector formulae*

$$A \equiv (\mathtt{x} \neq \mathtt{y}) \quad \textit{and} \quad \overline{B} \equiv (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)\,.$$

Table 3.7: Inference rules for EUF with transitive relations

$$\frac{t_1 = t_2}{t_2 = t_1}\ \textsf{Symm} \qquad \frac{t_1 > t_2}{t_1 \neq t_2}\ \textsf{DisEq} \qquad \frac{t_1 = t_2}{t_1 \geq t_2}\ \textsf{WeakenEq}$$

$$\frac{t_1 \geq t_2 \quad t_2 \geq t_1}{t_1 = t_2}\ \textsf{StrengthenInEq} \qquad \frac{t_1 = t_2}{f(t_1) = f(t_2)}\ \textsf{Cong}$$

$$\frac{t_1 \rhd_1 t_2 \qquad t_2 \rhd_2 t_3}{t_1 \rhd_3 t_3}\ \textsf{Trans} \qquad \text{with}$$

	$\rhd_3$	$=$	$\neq$	$\geq$	$>$
$\rhd_2$	$=$	$=$	$\neq$	$\geq$	$>$
	$\geq$	$\geq$		$\geq$	$>$
	$>$	$>$		$>$	$>$

(columns: $\rhd_1$; rows: $\rhd_2$)

$$\frac{t_1 \neq t_2 \qquad t_1 = t_2}{\textsf{false}}\ \textsf{Contra}_1 \qquad \frac{t_1 \geq t_2 \qquad t_2 > t_1}{\textsf{false}}\ \textsf{Contra}_2$$

The term $(\mathtt{z} \ll 1)$ *in* $\overline{B}$ *triggers the decision procedure to add to formula* $\overline{B}$ *the tautology* $(\mathtt{z} \ll 1) = \mathtt{z} + \mathtt{z}$ *(cf. Table 3.6(a)). The following proof is a refutation of the resulting conjunction of atoms:*

$$\dfrac{\dfrac{\dfrac{\mathtt{x} = \mathtt{z} \ll 1 \qquad \mathtt{z} \ll 1 = \mathtt{z} + \mathtt{z}}{\mathtt{x} = \mathtt{z} + \mathtt{z}}\ \textsf{Trans} \qquad \dfrac{\mathtt{y} = \mathtt{z} + \mathtt{z}}{\mathtt{z} + \mathtt{z} = \mathtt{y}}\ \textsf{Symm}}{\mathtt{x} = \mathtt{y}}\ \textsf{Trans} \qquad \mathtt{x} \neq \mathtt{y}}{\textsf{\textit{false}}}\ \textsf{Contra}_1$$

Since the set of rewriting and inference rules in the Tables 3.6(a) and 3.6(b) is incomplete, the algorithm may fail to refute a formula. In this case, we can fall back on the techniques described in Section 3.3 in order to obtain a complete interpolating decision procedure.

The formal definition of a proof resembles the definition of a resolution proof (Definition 3.3.2) but is slightly more general: It allows the application of general inference rules (Definition 3.2.1).

Definition 3.5.1. *A proof P in a theory $\mathcal{T}$ is a tree-shaped directed graph $(V_P, E_P, \ell_P, \mathrm{s}_P)$, where V_P is a set of vertices, E_P is a set of edges, ℓ_P is a function assigning formulae to vertices, and $\mathrm{s}_P \in V_P$ is a unique vertex (called the sink of P) with out-degree zero. An initial vertex has in-degree zero and is called a leaf of P. All other vertices are internal and have a non-zero in-degree. For each internal vertex $v \in V_P$ and its predecessor vertices*

$v_1, \ldots, v_n$ (where $(v_i, v) \in E_P$, $1 \leq i \leq n$) it holds that

$$\frac{\ell(v_1) \quad \cdots \quad \ell(v_n)}{\ell(v)},$$

*i.e., $\ell(v_1) \wedge \ldots \wedge \ell(v_n)$ entails $\ell(v)$ in $\mathcal{T}$. A proof P is a refutation if $\ell(\mathrm{s}_P) =$ **false**. A refutation is a refutation of a formula F if for each initial vertex $v \in V_P$ it holds that $\ell(v)$ is a conjunct of F.*

We emphasise that this definition is not restricted to the inference rules presented in this section. Moreover, the interpolation systems we discuss in the following can be applied to arbitrary local proofs (as introduced in Definition 3.2.2).

The following section shows how a reverse interpolant can be extracted from a local refutation of a pair of formulae.

3.5.2 Extracting Interpolants from Local Refutations

There is a variety of interpolation techniques for quantifier-free formulae in equality logic with uninterpreted functions. McMillan presents an interpolating inference system supporting equality logic with uninterpreted functions and linear arithmetic [McM05]. The decision procedure introduced in [FGG+09] constructs interpolants from congruence graphs representing EUF-formulae. In [KW09a], we present an approach which is similar to [FGG+09] but provides support for bit-vector operations and transitive relations by means of the inference rules presented in the previous section. Kovács and Voronkov [KV09b] presents a more general technique which enables the construction of interpolants from local proofs (defined in Section 3.2). To the best of our knowledge, there is no generalisation that subsumes the individual techniques (like labelled interpolations systems do for the propositional interpolation systems discussed in Section 3.3.3).

Despite the differences of the techniques listed above, the underlying principles are very similar. Intuitively, a reverse interpolant for the formulae $(A, \overline{B})$ represents assumptions over facts derived from $\overline{B}$ and provides guarantees about facts derived from A. If the assumptions about $\overline{B}$ hold, then so does the interpolant. Accordingly, the interpolant can be regarded as a form of rely-guarantee reasoning or as a contract between two players (as

suggested in [FGG+09]). The "facts" used to construct an interpolant can be extracted from a refutation $A, \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$.

Kovács and Voronkov [KV09b] shows that, given a local derivation $A, \overline{B} \vdash_{\mathcal{T}} C$ (Definition 3.2.2), there is a partial interpolant which is a Boolean combination of conclusions of the sub-proofs of the derivation. A partial interpolant may only refer to symbols shared by A and $\overline{B}$. Accordingly, the conclusions containing variables local to either A or $\overline{B}$ may not be used for the construction of an interpolant. For the remaining conclusions we need to decide whether to attribute them to A or to $\overline{B}$.

Definition 3.5.2 (Colour of Formulae/Vertices)**.** *Let $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ be a local refutation $A, \overline{B} \vdash_{\mathcal{T}} \mathsf{false}$ and let $C = \ell(v)$ (where $v \in V_P$) be a formula occurring in this proof either as a premise or as a conclusion.*

Assume that $\mathrm{Sym}(C) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$ *holds. Then we say that v is A-*coloured *if one of the following conditions holds:*

1. *v is a leaf of $A, B \vdash_{\mathcal{T}} \mathsf{false}$ and $A \vdash_{\mathcal{T}} C$, or*

2. *v is the conclusion of an inference step*

$$\frac{C_1 \quad \ldots \quad C_n}{C}$$

and at least one premise C_i $(1 \leq i \leq n)$ contains symbols local to A.

Otherwise, v is A-coloured if $\mathrm{Sym}(C) \cap (\mathrm{Sym}(A) \setminus \mathrm{Sym}(\overline{B})) \neq \emptyset$.

B-coloured vertices are defined analogously. If the vertex v corresponding to C is clear from the context, we also say that the formula *C is A-coloured (or B-coloured).*

In the terminology of [KV09b], an A-coloured formula using only shared symbols is a formula "justified by A". Note that Definition 3.5.2 leaves some freedom when it comes to colouring conclusions whose premises do not contain symbols local to A or $\overline{B}$. This is explicitly acknowledged in [FGG+09], leaving room for variation. The other algorithms consistently attribute such conclusions to B [McM05, KV09b] or do not provide specific instructions for this case [KW09a]. We assume from now on that such conclusions are coloured in an arbitrary but deterministic manner.

Given an A-coloured conclusion C in a proof $A, \overline{B} \vdash_{\mathcal{T}} \textsf{false}$, it is possible to determine a set of B-coloured formulae which, in conjunction with A, imply C. Intuitively, this set is the recursively closed set of B-coloured premises in the respective sub-proof. In accordance with [FGG+09, KW09a] we use B-premise(v) to denote this set. Given a single inference step $\ell(v_1), \ldots, \ell(v_n) \vdash_{\mathcal{T}} \ell(v)$ in a proof $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ (i.e., $(v_i, v) \in E_P$ for $1 \leq i \leq n$), we refer to the premises $\ell(v_1), \ldots, \ell(v_n)$ as *direct* premises of $\ell(v)$ in order to avoid confusion.

Definition 3.5.3 (A-Premises, B-Premises)**.** *Let $\ell(v_1), \ldots, \ell(v_n) \vdash_{\mathcal{T}} \ell(v)$ be an inference step in a local refutation $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ representing $A, \overline{B} \vdash_{\mathcal{T}}$ **false** (i.e., $(v_i, v) \in E_P$, $1 \leq i \leq n$). Assume $v \in V_P$ is a B-coloured vertex. Then the A-premise of v is defined as*

$$A\text{-}premise\,(v) \stackrel{\text{def}}{=}$$

$$\{\ell(v_i) \,|\, (v_i, v) \in E_P \text{ and } v_i \text{ is } A\text{-coloured}\} \cup$$

$$\bigcup\{A\text{-}premise\,(v_i) \,|\, (v_i, v) \in E_P \text{ and } v_i \text{ is } B\text{-coloured}\}\,.$$

For an A-coloured conclusion $\ell(v)$, the set B-premise(v) is defined analogously.

Lemma 1 in [KV09b] states that given an A-coloured conclusion $\ell(v)$ in a local refutation $A, \overline{B} \vdash_{\mathcal{T}} \textsf{false}$ it holds that A, B-premise$(v) \vdash_{\mathcal{T}} \ell(v)$. This follows immediately from the structural restrictions for local proofs. For a formal justification we refer the reader to [KV09b, Lemma 1] and [FGG+09, Lemma 3]. We remark briefly that A-premises take the role of ρ in McMillan's interpolations [McM05] and B-premises correspond to justifications in [FGG+09]. A detailed discussion of this aspect is provided in [FGG+09, Section 6].

We are now in a position to provide an inductive definition of partial interpolants for local refutation proofs $A, \overline{B} \vdash_{\mathcal{T}} \textsf{false}$. We start by refining our definition of interpolation systems (Definition 3.2.5) according to the current setting.

Definition 3.5.4 (Interpolation System for Local Refutations)**.** *An interpolation system for a local refutation P of a pair of formulae $(A, \overline{B})$ is a function mapping vertices $v \in V_P$ for which* $\mathrm{Sym}(\ell(v)) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$ *holds to partial interpolants. Given a refutation P representing $A, \overline{B} \vdash_{\mathcal{T}}$ **false** we use $\mathsf{Itp}^{\mathcal{T}}(P, A, \overline{B})$ to denote the partial mapping from V_P to partial interpolants.*

An interpolation system $\mathsf{Itp}^{\mathcal{T}}$ *for local refutations is correct if for every refutation* P *of* $A \wedge \overline{B}$ *it holds that* $\mathsf{Itp}^{\mathcal{T}}(P, A, \overline{B})(\mathrm{s}_P)$ *(where* $\ell(\mathrm{s}_P) = \textsf{false}$*) is a Craig interpolant for the pair of formulae* (A, B)*.*

The following definition introduces the interpolation system $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}$ for local refutations in a theory $\mathcal{T}$ presented by Kovács and Voronkov in [KV09b].

Definition 3.5.5. *Let* $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ *be a local refutation of the conjunction of the pair of formulae* $(A, \overline{B})$*. The interpolation system* $\mathsf{Itp}^{\mathcal{T}}_{KV}$ *maps vertices* $v \in V_P$ *for which* $\mathrm{Sym}(\ell(v)) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$ *holds to partial interpolants according to the following rules:*

1. *For each vertex* v *such that* $\ell(v)$ *is derived from either* A *or from* $\overline{B}$ *in* P *let*

$$\mathsf{Itp}^{\mathcal{T}}_{KV}(P, A, \overline{B})(v) = \begin{cases} \ell(v) & \text{if } A \vdash_{\mathcal{T}} \ell(v) \text{ in } P \\ \neg \ell(v) & \text{if } \overline{B} \vdash_{\mathcal{T}} \ell(v) \text{ in } P \end{cases} .$$

2. *Otherwise,* v *is an internal vertex. We use* $\{C_1, \ldots, C_n\}$ *to denote the elements of the set* B*-premise*(v) *if* v *is* A*-coloured and* A*-premise*(v) *if* v *is* B*-coloured, respectively. Furthermore, we use* $I_1, \ldots, I_n$ *to denote the respective partial interpolants of* $C_1, \ldots, C_n$*. Then*

$$\mathsf{Itp}^{\mathcal{T}}_{KV}(P, A, \overline{B})(v) = \begin{cases} \bigwedge_{i=1}^{n} (C_i \vee I_i) \wedge \neg \bigwedge_{i=1}^{n} C_i & \text{if } v \text{ is } A\text{-coloured} \\ \bigwedge_{i=1}^{n} (C_i \vee I_i) & \text{if } v \text{ is } B\text{-coloured} \end{cases} .$$

A proof of correctness for the interpolation system introduced in Definition 3.5.5 is provided in [KV09b].

In the following, we introduce an alternative interpolation system. The interpolation system $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}$ presented in the following definition is similar[4] to the interpolation system presented in our paper [KW09a], where we provide an algorithmic description of our interpolation technique.

[4] The fact that interpolants are not defined inductively in [KW09a] allows us to construct formulae of a structure that contains less nesting.

Definition 3.5.6. *Let $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ be a local refutation of the conjunction of the pair of formulae $(A, \overline{B})$. The interpolation system $\mathsf{Itp}^{\mathcal{T}}_{KW}$ maps vertices $v \in V_P$ for which $\mathrm{Sym}(\ell(v)) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$ holds to partial interpolants according to the following rules:*

1. *For each vertex v such that $\ell(v)$ is derived from either A or from $\overline{B}$ in P let*

$$\mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(v) = \begin{cases} \ell(v) & \text{if } A \vdash_{\mathcal{T}} \ell(v) \text{ in } P \\ \neg\ell(v) & \text{if } \overline{B} \vdash_{\mathcal{T}} \ell(v) \text{ in } P \end{cases} .$$

2. *Otherwise, v is an internal vertex. We use $\{C_1, \ldots, C_n\}$ to denote the elements of the set B-premise(v) if v is A-coloured and A-premise(v) if v is B-coloured, respectively. Furthermore, we use $I_1, \ldots, I_n$ to denote the respective partial interpolants of $C_1, \ldots, C_n$. Then*

$$\mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(v) = \begin{cases} \bigvee_{i=1}^{n} (\neg C_i \wedge I_i) & \text{if } v \text{ is } A\text{-coloured} \\ \bigvee_{i=1}^{n} (\neg C_i \wedge I_i) \vee \bigwedge_{i=1}^{n} C_i & \text{if } v \text{ is } B\text{-coloured} \end{cases} .$$

Theorem 3.5.1 (Correctness of $\mathsf{Itp}^{\mathcal{T}}_{KW}$)**.** *For any local refutation P of a formula $A \wedge \overline{B}$, $\mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(\mathrm{s}_P)$ is a reverse interpolant for the pair of formulae $(A, \overline{B})$.*

Before we provide a proof of Theorem 3.5.1 we present the following auxiliary result.

Lemma 3.5.1. *Let P be a local refutation of the conjunction of a pair of formulae $(A, \overline{B})$ and let $v \in V_P$ be a vertex such that v is A-coloured (B-coloured, respectively). Let $\{C_1, \ldots, C_n\}$ denote the set B-premise(v) (A-premise(v), respectively). Then it holds for all C_i, $1 \leq i \leq n$, that $\mathrm{Sym}(C_i) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$.*

Proof. W.l.o.g., we assume that v is B-coloured (Definition 3.5.2). Since P is a local proof (Definition 3.2.2), it must hold for all predecessor vertices $v_i \in V_P$ (i.e., $(v_i, v) \in E_P$) that $\mathrm{Sym}(\ell(v_i)) \subseteq \mathrm{Sym}(B)$. Accordingly, it must holds for all A-coloured formulae C in $\{\ell(v_1), \ldots, \ell(v_n)\}$ that $\mathrm{Sym}(C) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$. It follows immediately from the inductive definition of the A-premise (Definition 3.5.3) that the same holds for all formulae in the set A-premise(v). □

The following proof establishes the correctness of Theorem 3.5.1.

Proof. We prove Theorem 3.5.1 by using induction over the structure of P. Note that, if $\{\ell(v_1),\ldots,\ell(v_n)\}$ denotes the set A-premise(v) (or B-premise(v), respectively), then the vertices $v_1,\ldots,v_n$ are (not necessarily direct) predecessors of the vertex $v \in V_P$. Let $v \in V_P$ be a vertex such that $C = \ell(v)$, $\mathrm{Sym}(C) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$, and $I = \mathsf{Itp}^{\mathcal{T}}(P, A, \overline{B})(v)$. We show that I is a partial interpolant for v, i.e., the following conditions hold:

1. $A \wedge \neg C \models_{\mathcal{T}} I$,
2. $\overline{B} \wedge \neg C \models_{\mathcal{T}} \neg I$, and
3. $\mathrm{Sym}(C) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$.

Base case. Let $v \in V_P$ be a vertex in the proof P and let $C = \ell(v)$. Suppose that $\ell(v)$ is derived from either A or $\overline{B}$ in P (i.e., all leaves of the sub-tree rooted at v are either labelled with premises from A or with premises from $\overline{B}$). We need to distinguish two cases. If $A \vdash_{\mathcal{T}} C$ and $I = C$, then $A \wedge \neg C \models_{\mathcal{T}} I$ as well as $\overline{B} \wedge \neg C \models_{\mathcal{T}} \neg I$ hold trivially, and the third condition holds by definition of v. The case in which $\overline{B} \vdash_{\mathcal{T}} C$ and $I = \neg C$ is symmetric.

Induction hypothesis: Let $v \in V_P$ be a vertex in P. Suppose that v is a B-coloured (A-coloured, respectively) formula. Let $\{C_1,\ldots,C_n\}$ denote the set A-premise(v) (or the set A-premise(v), respectively) and $I_1,\ldots,I_n$ the respective partial interpolants. Then it holds for all C_i ($1 \le i \le n$) that $A \wedge \neg C_i \models_{\mathcal{T}} I_i$, $\overline{B} \wedge \neg C_i \models_{\mathcal{T}} \neg I_i$, and $\mathrm{Sym}(I_i) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$.

Induction step. Let $v \in V_P$ be an internal vertex and let $C = \ell(v)$. Furthermore, let $\{C_1,\ldots,C_n\}$ denote the set B-premise(v) if v is A-coloured and A-premise(v) otherwise, and let $I_1,\ldots,I_n$ denote the respective partial interpolants. We need to distinguish two cases.

1. The vertex v is A-coloured and $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v) = \bigvee_{i=1}^{n}(\neg C_i \wedge I_i)$. It follows from A, B-premise(v) $\models_{\mathcal{T}} C$ that $\bigvee_{i=1}^{n}(A \wedge \neg C \models_{\mathcal{T}} \neg C_i)$ holds. Using the induction hy-

pothesis $A \wedge \neg C_i \models_{\mathcal{T}} I_i$, we derive $\bigvee_{i=1}^{n}(A \wedge \neg C \models_{\mathcal{T}} I_i)$. If we conjoin these results, we obtain $\bigvee_{i=1}^{n}(A \wedge \neg C \models_{\mathcal{T}} \neg C_i \wedge I_i)$ and subsequently $A \wedge \neg C \models_{\mathcal{T}} \bigvee_{i=1}^{n} \neg C_i \wedge I_i$.

For the second condition, we need to show that $\overline{B} \wedge \neg C \models_{\mathcal{T}} \bigwedge_{i=1}^{n}(C_i \vee \neg I_i)$, i.e., that $\overline{B} \wedge \neg C \models_{\mathcal{T}} (C_i \vee \neg I_i)$ holds for all $i \in \{1..n\}$. This follows immediately from the induction hypothesis $\overline{B} \wedge \neg C_i \models_{\mathcal{T}} \neg I_i$.

The third condition follows immediately from the induction hypothesis and from Lemma 3.5.1.

2. The vertex v is B-coloured and $\mathsf{Itp}_{\mathrm{KW}}^{\mathcal{T}}(P, A, \overline{B})(v) = \bigvee_{i=1}^{n}(\neg C_i \wedge I_i) \vee \bigwedge_{i=1}^{n} C_i$. It follows from $\overline{B}, A\text{-premise}(v) \models_{\mathcal{T}} C$ that $B \wedge \neg C \models_{\mathcal{T}} \neg \bigwedge_{i=1}^{n} C_i$ holds. Moreover, from the induction hypothesis $\overline{B} \wedge \neg C_i \models_{\mathcal{T}} \neg I_i$ (where $1 \leq i \leq n$) we derive that $\overline{B} \models_{\mathcal{T}} \bigwedge_{i=1}^{n}(C_i \vee \neg I_i)$. In conjunction with the previous result, this yields $\overline{B} \wedge \neg C \models_{\mathcal{T}} \bigwedge_{i=1}^{n}(C_i \vee \neg I_i) \wedge \neg \bigwedge_{i=1}^{n} C_i$ and subsequently $\overline{B} \wedge \neg C \models_{\mathcal{T}} \neg(\bigvee_{i=1}^{n}(\neg C_i \wedge I_i) \vee \bigwedge_{i=1}^{n} C_i)$.

   Note that the first condition $A \wedge \neg C \models_{\mathcal{T}} \bigvee_{i=1}^{n}(\neg C_i \wedge I_i) \vee \bigwedge_{i=1}^{n} C_i$ holds if the conjunction $\bigwedge_{i=1}^{n} C_i$ holds. Otherwise, at least one C_i ($i \in \{1..n\}$) must evaluate to false. But then I_i must hold as well according to the induction hypothesis $A \wedge \neg C_i \models_{\mathcal{T}} I_i$. Accordingly, $\neg C_i \wedge I_i$ holds. This establishes that the first condition must hold, too.

   The third condition follows again from the induction hypothesis and from Lemma 3.5.1.

□

The interpolation systems introduced in Definitions 3.5.5 and 3.5.6 differ in so far as they map internal vertices to different partial interpolants. The following theorem states that, similar to the propositional interpolation systems $\mathsf{Itp}_{\mathrm{M}}$ and $\mathsf{Itp}_{\mathrm{HKP}}$ (see Section 3.3.4), the interpolants generated by $\mathsf{Itp}_{\mathrm{KV}}^{\mathcal{T}}(P, A, \overline{B})$ are logically stronger than the interpolants obtained using $\mathsf{Itp}_{\mathrm{KW}}^{\mathcal{T}}(P, A, \overline{B})$.

Theorem 3.5.2. *Let $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ be a local refutation of the conjunction of the pair of formulae $(A, \overline{B})$. Furthermore, assume an arbitrary but fixed colouring of the vertices. Then $\mathsf{Itp}_{KV}^{\mathcal{T}}(P, A, \overline{B})(v) \Rightarrow \mathsf{Itp}_{KW}^{\mathcal{T}}(P, A, \overline{B})(v)$ holds for each vertex $v \in V_P$ for which $\mathrm{Sym}(\ell(v)) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(\overline{B})$ holds.*

A proof of this theorem is provided in Appendix B (Section B.5). The following example shows that there are proofs for which the interpolation systems $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})$ and $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})$ yield different interpolants.

Example 3.5.2. *Consider the formulae*

$$\begin{aligned} A &\equiv (\mathtt{x} = \mathtt{y}) \wedge (\mathtt{u} = \mathtt{v}) \wedge (\mathtt{x} = \mathtt{m}) \wedge (\mathtt{n} = \mathtt{v}) \qquad \textit{and} \\ \overline{B} &\equiv (\mathtt{y} = \mathtt{u}) \wedge (\mathtt{m} \neq \mathtt{n}) \wedge (\mathtt{x} \leq \mathtt{v})\,. \end{aligned}$$

In the following annotated local refutation of the conjunction $A \wedge \overline{B}$ the atoms in grey represent B-coloured formulae. Note that for all inner vertices we have the choice to colour them either A or $\overline{B}$, since all symbols are shared. The annotations generated by the interpolation systems are shown in square brackets. To improve the readability, we only show the partial interpolants that are required to construct the final interpolant.

$$\dfrac{\dfrac{\dfrac{\mathtt{x} = \mathtt{y}\ [\mathtt{x} = \mathtt{y}] \qquad \mathtt{y} = \mathtt{u}}{\mathtt{x} = \mathtt{u}\ [\mathtt{x} = \mathtt{y}]} \qquad \mathtt{u} = \mathtt{v}}{\mathtt{x} = \mathtt{v}} \qquad \dfrac{\dfrac{\mathtt{x} = \mathtt{m}\ [\mathtt{x} = \mathtt{m}] \qquad \mathtt{m} \neq \mathtt{n}}{\mathtt{x} \neq \mathtt{n}\ [\mathtt{x} = \mathtt{m}]} \qquad \mathtt{n} = \mathtt{v}}{\mathtt{x} \neq \mathtt{v}}}{\textsf{false} \quad [\textit{see text}]}$$

Both interpolation systems yield the same annotations for the initial vertices of the proof and for all internal vertices except for the sink. We obtain the following interpolants:

$$\begin{aligned} \mathsf{Itp}^{\mathcal{T}}_{KV}(P, A, \overline{B})(\mathrm{s}_P) &= (\mathtt{x} = \mathtt{u} \vee \mathtt{x} = \mathtt{y}) \wedge (\mathtt{x} \neq \mathtt{n} \vee \mathtt{x} = \mathtt{m}) \wedge (\mathtt{x} \neq \mathtt{u} \vee \mathtt{x} = \mathtt{n}) \\ \mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(\mathrm{s}_P) &= (\mathtt{x} \neq \mathtt{u} \wedge \mathtt{x} = \mathtt{y}) \vee (\mathtt{x} = \mathtt{n} \wedge \mathtt{x} = \mathtt{m}) \end{aligned}$$

Now consider a valuation to the symbols which makes the formulae $\mathtt{x} = \mathtt{n}$, $\mathtt{x} \neq \mathtt{m}$, $\mathtt{x} \neq \mathtt{u}$, and $\mathtt{x} = \mathtt{y}$ evaluate to **true**. *Then $\mathsf{Itp}^{\mathcal{T}}_{KV}(P, A, \overline{B})(\mathrm{s}_P)$ evaluates to* **false** *and $\mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(\mathrm{s}_P)$ evaluates to* **true**, *demonstrating that the interpolants are not logically equivalent.*

A more detailed comparison of all interpolation systems [McM05, FGG+09, KV09b, KW09a] exceeds the scope of this dissertation. We conclude the section revisiting two examples we encountered in the Examples 2.2.2 and 3.5.1.

Example 3.5.3. *Consider the formulae* $A \equiv (\mathtt{x} \neq \mathtt{y})$ *and* $\overline{B} \equiv (\mathrm{y} = \mathrm{z} + \mathrm{z}) \wedge (\mathrm{x} = \mathrm{z} \ll 1)$ *previously presented in Example 3.5.1 in this section. In the following refutation for* $A \wedge \overline{B}$ *we use the same colour coding as in Example 3.5.2:*

$$\dfrac{\dfrac{\dfrac{\mathrm{x} = \mathrm{z} \ll 1 \qquad \mathrm{z} \ll 1 = \mathrm{z} + \mathrm{z}}{\mathrm{x} = \mathrm{z} + \mathrm{z}}\,\text{Trans} \qquad \dfrac{\mathrm{y} = \mathrm{z} + \mathrm{z}}{\mathrm{z} + \mathrm{z} = \mathrm{y}}\,\text{Symm}}{\mathrm{x} = \mathrm{y} \quad [\mathtt{x} \neq \mathtt{y}]} \qquad \mathtt{x} \neq \mathtt{y}\ [\mathtt{x} \neq \mathtt{y}]}{\textit{false} \quad [\mathtt{x} \neq \mathtt{y}]}\,\text{Contra}_1$$

As expected, we obtain the interpolant $\mathtt{x} \neq \mathtt{y}$*.*

Example 3.5.4. *Finally, we revisit the example presented in our introduction (Section 1.3.1). A local refutation for the conjunction of the formulae* $A \equiv (\mathtt{y} = \mathtt{x}) \wedge (\mathtt{y} \neq 0)$ *and* $\overline{B} \equiv (\mathtt{y}' = \mathtt{y}\&(\mathtt{y} - 1)) \wedge (\mathtt{x} = \mathtt{y}')$ *is presented in Example 2.2.2:*

$$\dfrac{\dfrac{\dfrac{\mathtt{y} = \mathtt{x}}{\boxed{\mathtt{y} \leq \mathtt{x}}} \qquad \dfrac{\boxed{\mathtt{y} \neq 0} \qquad \dfrac{\mathrm{y}' = \mathrm{y}\&(\mathrm{y} - 1)}{\mathrm{y}' \leq \mathrm{y} - 1}}{\mathrm{y}' < \mathrm{y}}}{\mathrm{y}' < \mathrm{x}} \qquad \mathrm{x} = \mathrm{y}'}{\textit{false}}$$

The A-premise of the B-coloured conclusion **false** *is marked in the proof. Both interpolation systems yield the interpolant* $(\mathtt{y} \neq 0) \wedge (\mathtt{y} \leq \mathtt{x})$*.*

The following section describes the experimental results obtained by integrating our interpolating decision procedure (described in more detail in Section A.2) into our interpolation-based verification tool Wolverine.

3.6 Experimental Results

The simplified Windows device drivers presented in [HJMS02, McM06] have become the baseline benchmark for software verification tools. In this section, we present an evaluation of the performance of Wolverine on this benchmark. Wolverine is an implementation of the interpolation-based verification algorithm presented in Section 2.5 and uses the refinement approach of Section 2.5.3. Since the implementation of McMillan's interpolation-based

Table 3.8: Verification of the simplified Windows device drivers from [HJMS02, McM06] and [BCG+09]. The performance is measured in seconds.

device driver	WOLVERINE lifting	WOLVERINE no lifting	BLAST 2.5 DFS	BLAST 2.5 BFS
`kbfiltr.i`	00:01.84	00:02.18	00:02.22	00:04.54
`cdaudio.i`[6]	01:03.54	01:27.95	00:54.48	error
`diskperf.i`	01:11.80	06:17.63	06:14.41	00:40.03
`floppy.i`	02:02.36	03:16.22	14:18.92	01:40.05
`parport.i`	7:31.87	21:27.13	02:22:16	05:32.74

verification tool IMPACT [McM06] is not available, we provide the run-time of the program verifier BLAST as a reference point. Note that, unlike WOLVERINE, BLAST performs a predicate abstraction-based image computation. Moreover, BLAST is a more mature tool featuring a variety of optimisations not present in WOLVERINE.[5] We abstain from performing a comparison with the BLAST-based verification tool CPACHECKER presented in [BCG+09], since its algorithm differs significantly from the path-based search algorithm discussed in Section 2.5. Notably, [BCG+09] reports a significant improvement over the stand-alone version of BLAST. We emphasise, however, that the optimisations suggested in [BCG+09] can also be implemented in WOLVERINE.

We obtained a binary release of BLAST version 2.5 from the distribution provided by [BCG+09].[7] As suggested in [McM06], we ran the BLAST executable `pblast.opt` with the parameters `-msvc -nofp -tproj -cldepth 1 -predH 6 -scope -lolattice -clock -dfs` (performing a depth-first search) and a second time with `-msvc -nofp -craig 2 -scope -cldept 1 -bfs` (running a breadth-first search). We ran all our experiments on a 3 GHz Intel Xeon quad-core machine with 48 GB of memory. The verification tools we evaluate are not parallelised and therefore only able to use a single core. Moreover, the tools used less than 5% of the available memory. The performance results of these experiments are presented Table 3.8. The verification of the device driver `cdaudio.i` using breadth-first search terminated with an error message of BLAST (adding the parameter `-clock`, as sug-

[5]Personal communication with Dirk Beyer, who calls BLAST a "hybrid, highly-tuned monster".

[6]The verification of this driver required the insertion of the code fragment `IRP irp; pirp=&irp;` at the beginning of the `main` function, since BLAST and the symbolic simulator of the CPROVER framework (which adheres to the ANSI-C standard) differ in their interpretation of uninitialised global pointers and data structures.

[7]We failed to compile or run the official version which is available from `http://mtc.epfl.ch/software-tools/blast/index-epfl.php`.

gested in [McM06], did not change this outcome). Furthermore, the version of BLAST we used is unable to parse the `parclass.i` device driver. This is consistent with the results reported in [BCG+09]. While the parser of the CPROVER framework is capable of processing the file `parclass.i`, our decision procedure fails on this example because of an error in the current version of the implementation. Therefore, we refrain from listing the run-time for `parclass.i`.

We evaluate the performance of our verification tool WOLVERINE with and without the proof-lifting technique presented in Section 3.4 enabled. In the former case, WOLVERINE uses the bit-flattening method described in Section 3.3.1 to compute an unsatisfiable core and the lifting technique of Section 3.4.2 to derive blocking clauses. These blocking clauses serve as an interface to the proof-generating decision procedure in Appendix A.2. The interpolants are generated using a combination of the propositional interpolation system $\mathsf{Itp}_{\mathrm{M}}$ (see Section 3.3.3) and a variant (described in [KW09a]) of the word-level interpolation system $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}$ introduced in Section 3.5. WOLVERINE uses this interpolation technique by default. The column labelled "lifting" in Table 3.8 lists the results of running WOLVERINE with the parameters `--error-label ERROR --no-library --32 --i386-win32`. To obtain the results in the column labelled "no lifting", we deactivated the proof-lifting technique using the command line parameter `--no-proof-lifting`. In this case, WOLVERINE falls back to the basic SMT approach[8] presented in Section 3.4. The results in Table 3.8 show that proof-lifting results in a consistently superior performance of the verification tool.

BLAST, in contrast, relies on FOCI, an implementation of the interpolating decision procedure presented in [McM05]. Since FOCI (as well as other available interpolation tools such as CSISAT [BZM08] and MATHSAT4 [BCF+08]) approximates the bit-vector semantics of programs using linear inequalities over the reals, a comparison with our interpolation system has limited informative value. While WOLVERINE is not faster than BLAST, the performance difference is not large. We attribute the advantage of BLAST to its maturity. We defer additional optimisations of WOLVERINE's model checking algorithm to future work.

Notably, the inference rules presented in Section 3.5 in combination with a small number

[8]Our basic SMT algorithm does not feature elaborate optimisations such as the ones described in [CGS10].

of additional rules dealing with arrays (cf. Section A.2 in the appendix) and typecasts (an artefact commonly encountered in ANSI-C programs) suffices to verify the simplified Windows device drivers listed in Table 3.8. In order to corroborate this observation, we use Wolverine to analyse the (unmodified) `machzwd.c` device driver for Linux provided as part of our device driver verification framework DDVerify [WBKW07].

The DDVerify framework provides a simplified model of the operating system in order to separate the verification of the operating system from the task of model checking the device driver. Furthermore, given the source code of a Linux device driver, DDVerify automatically generates a test-harness which contains calls to the functions of the driver (in non-deterministic order). The original version of DDVerify (presented in [WBKW07]) uses the software model checker SatAbs [CKSY05]. SatAbs is a predicate-abstraction based verification tool which generates Boolean programs (see Section 2.4.3) and uses the `SMV` model checker [McM92] to check the abstraction. Accordingly, SatAbs uses an eager refinement strategy. A more recent version [KW09b] of DDVerify (available on `http://www.cprover.org/ddverify`) also provides support for other verification tools such as Cbmc [CKL04] and Wolverine.

The operating system model of DDVerify contains a number of assertions. These assertions ensure, for instance, that resources are not used unless they have been allocated. In particular, the option `--check-io` instructs DDVerify to generate assertions for verifying the correct usage of IO ports. We ran `ddverify machzwd.c --check-io --model seq1` to generate a sequential harness for the `machzwd.c` device driver provided with DDVerify. This results in the verification conditions (claims) listed in Table 3.9. For clarity, Figure 3.25 illustrates these very results using a bar diagram. We used Wolverine and SatAbs to check whether these claims hold. As reported in [WBKW07], the claims `zf_readw.1` and `zf_readw.2` do not hold, since the driver accesses unallocated IO resources in its initialisation phase in an attempt to detect the presence of its related hardware logic. The remaining claims listed in the lower section of Table 3.9 hold.

The fact that Wolverine avoids the use of predicate abstraction to construct an abstract transition function gives it an advantage over SatAbs in the cases where the claims do not hold (`zf_readw.1` and `zf_readw.2`). This observation, however, cannot be gen-

Table 3.9: Performance results (in seconds) for WOLVERINE and SATABS on the `machzwd.c` Linux device driver presented in [WBKW07]

Claim	WOLVERINE	SATABS
`zf_readw.1`	00:02.60	00:33.44
`zf_readw.2`	00:02.56	00:35.04
`zf_set_control.1`	00:51.63	00:16.15
`zf_set_control.2`	00:51.15	00:17.17
`zf_set_control.3`	00:05.96	01:43.55
`zf_set_control.4`	00:05.88	01:44.39
`zf_set_control.5`	00:21.97	00:26.14
`zf_set_control.6`	error	00:26.64
`zf_set_control.7`	00:21.96	00:26.62
`zf_set_control.8`	error	00:26.97
`zf_set_control.9`	00:29.92	00:06.96
`zf_set_control.10`	00:29.73	00:06.96
`zf_timer_on.1`	00:06.74	00:30.36
`zf_timer_on.2`	00:06.73	00:30.83
`zf_set_timer.1`	00:07.41	00:36.70
`zf_set_timer.2`	00:07.48	00:30.34
`zf_set_timer.3`	00:07.41	00:40.63
`zf_set_timer.4`	00:07.44	00:35.21
`zf_ping.1`	00:07.67	00:17.33
`zf_ping.2`	error	00:17.52
`zf_set_status.1`	00:27.21	00:06.34
`zf_set_status.2`	00:28.37	00:06.35

eralised to the other experiments. In the case of claim `zf_set_control.1`, for instance, SATABS' eager refinement approach pays off – WOLVERINE repeatedly derives the same interpolants in different branches of the reachability tree, while SATABS needs to discover the corresponding predicates only once. In contrast, the picture is reversed for the claim `zf_set_control.3`. The driver harness generated by DDVERIFY contains a `case`-construct discriminating between 26 different calls to the functions of the driver, only one of which is relevant to the claim. SATABS generates 72 global predicates to model this case split, thus putting significant strain on the SMV model checker. WOLVERINE, on the other hand, is able to *locally* discharge the irrelevant branches in the reachability tree. This observation shows that neither an eager nor a lazy refinement strategy is consistently superior (cf. Section 2.5.3).

During the verification of the `machzwc.c` driver, we encounter three cases (namely the claims `zf_set_control.6`, `zf_set_control.8`, and `zf_ping.2`) in which our decision pro-

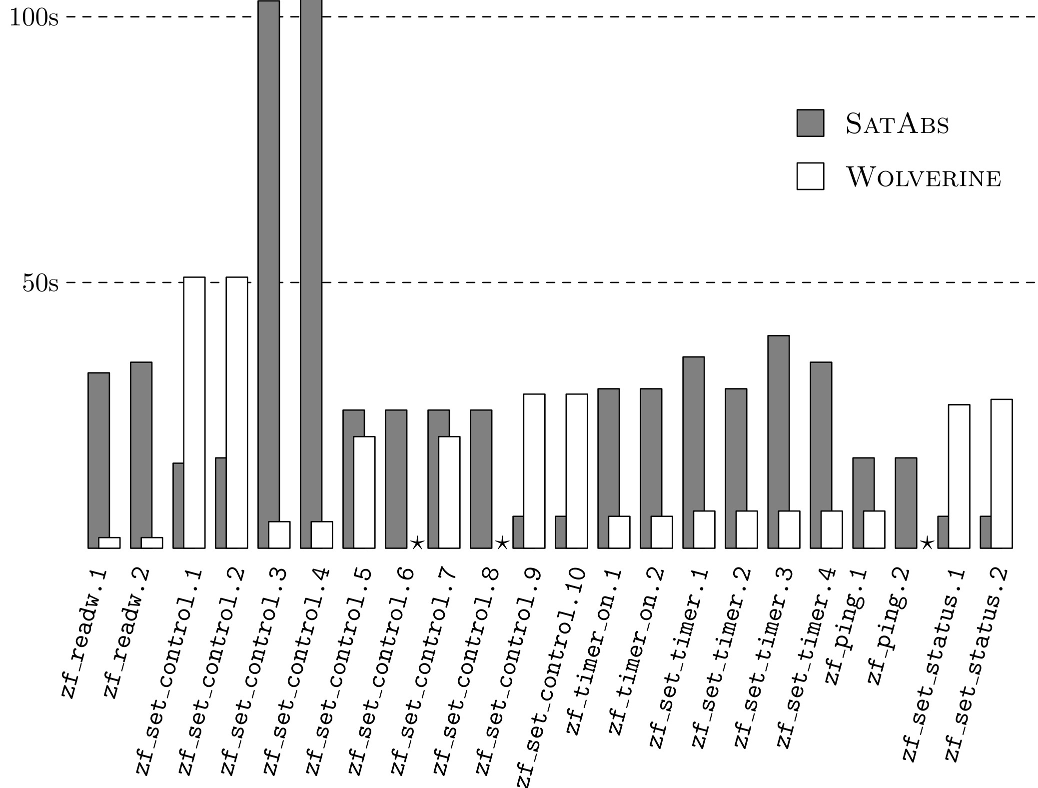


Figure 3.25: Results from Table 3.9 presented using a bar diagram

cedure fails to derive an interpolant from a spurious counterexample. In these three cases, our inference rules dealing with arrays and compound data structures are not sufficient. We intend to eliminate this shortcoming in a future version of Wolverine. SatAbs, in contrast, derives its refinement predicates using the weakest precondition, which is a much less complex operation than our interpolation algorithm. In a future version of Wolverine, we plan to implement a strategy that falls back on the weakest precondition to compute interpolants in case the interpolating decision procedure fails.

We conclude the section with a preliminary evaluation of the impact of the strength of interpolants. To this end, we compare the performance results of running Wolverine on the `machzwd.c` driver, once using McMillan's interpolation system $\mathsf{ltp}_{\mathrm{M}}(R, A, \overline{B})$ and a second time using its weaker inverse counterpart $\neg\mathsf{ltp}_{\mathrm{M}}(R, \overline{B}, A)$ instead (cf. Lemma 3.2.2 and Theorem 3.3.5). By default, Wolverine uses the stronger interpolation system. The weaker version is activated using the parameter `--weak-interpolants`. Table 3.10 shows the result of this experiment (also illustrated by Figure 3.26).

Table 3.10: Performance results (in seconds) for WOLVERINE with strong and weak interpolants on the `machzwd.c` Linux device driver

Claim	strong	weak
`zf_set_control.1`	00:51.63	00:13.49
`zf_set_control.2`	00:51.15	00:13.59
`zf_set_control.3`	00:05.96	00:06.25
`zf_set_control.4`	00:05.88	00:06.27
`zf_set_control.5`	00:21.97	00:02.41
`zf_set_control.7`	00:21.96	00:02.15
`zf_set_control.9`	00:29.92	00:01.83
`zf_set_control.10`	00:29.73	00:01.82
`zf_timer_on.1`	00:06.74	00:06.78
`zf_timer_on.2`	00:06.73	00:06.76
`zf_set_timer.1`	00:07.41	00:07.45
`zf_set_timer.2`	00:07.48	00:07.48
`zf_set_timer.3`	00:07.41	00:07.47
`zf_set_timer.4`	00:07.44	00:07.56
`zf_ping.1`	00:07.67	00:02.26
`zf_set_status.1`	00:27.21	00:01.80
`zf_set_status.2`	00:28.37	00:01.84

In the cases where interpolant strength makes a difference (`zf_set_control.`n, with $n \in \{1, 2, 5, 7, 9, 10\}$, and `zf_set_status.1` and `zf_set_status.2`) the results in Table 3.10 paint a picture in favour of weak interpolants. In order to verify claim `zf_set_control.1`, for instance, WOLVERINE explores 10^3 nodes of the reachability tree when using weak interpolants, and 50% more when using strong interpolants. A cursory analysis shows that the cause of this phenomenon is not so much the strength of the interpolants, but rather the fact that the labelling functions L_M and $L_{M'}$ (introduced in Lemma 3.3.1) result in a different partitioning of the blocking clauses (see Lemma 3.4.1). This forces $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}$ to use different conclusions from the word-level refutation to construct the respective partial interpolants (see Definition 3.5.6). Specifically, the labelling $L_{M'}$ assigns the label a to shared skeleton literals, resulting in blocking formulae with a "larger" A-partition. Consequently, if we colour the final conclusion of word-level refutations with "B" (as done in Example 3.5.4 and our implementation), the a-labelled theory-atoms are "more likely" to end up in the interpolant generated by $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}$.

This delicate interaction of the propositional and the word-level interpolation system has the side-effect that the word-level interpolants generated by combining $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}$ with either

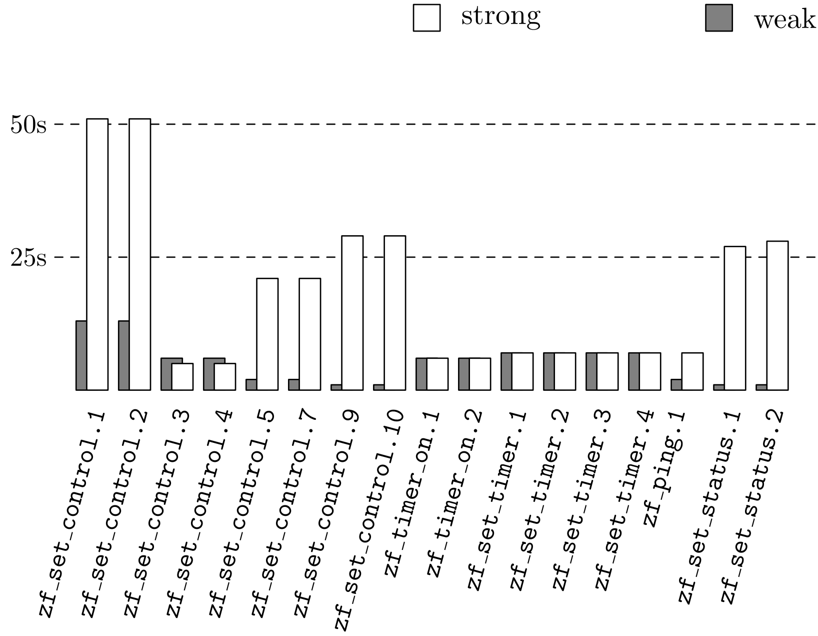


Figure 3.26: Results from Table 3.10 presented using a bar diagram

$\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})$ or $\neg\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)$ are *not* necessarily ordered by strength. This observation is also confirmed by our experiments.

Furthermore, to conclude that weak interpolants are always superior would be premature. In fact, Wolverine in combination with the weak interpolation system fails to find an invariant for the simplified device drivers listed in Table 3.8 (except for `kbfiltr.i`, where it performs slightly worse than Wolverine without proof lifting). In all cases, we observe interpolants with a highly disjunctive structure that are significantly larger than the ones obtained by means of $\mathsf{Itp}_{\mathrm{M}}$. We defer detailed investigation of this problem to future work.

The conclusion that we can draw so far is that the verification techniques presented in Chapter 2 are very sensitive to changes of the interpolation algorithm. This insight is neither new nor particularly surprising: it is well known that the performance of predicate abstraction-based verification tools is extremely contingent on the refinement algorithm [HJMM04]. We conjecture that an interpolation algorithm which carefully includes or avoids certain theory-atoms (enabled by Theorem 3.3.1) may have a larger impact on the performance than interpolant strength. A further exploration of this idea, however, is subject to future work.

3.7 Related Work

The related work in the area of decision procedures is vast. We focus on recent *interpolating* decision procedures. The first implementation of an interpolating decision procedure widely used in verification is McMillan's FOCI [McM05]. This tool supports linear inequalities over $\mathbb{R}$ and equality with uninterpreted functions (EUF), and introduces the semantic discrepancy discussed in Example 3.1.1 when used for program verification. Beyer et al. [BZM08] presents an interpolating decision procedure for the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols. Based on the ideas in [McM05], Fuchs presents a graph-based approach for EUF [FGG+09]. The interpolants in CNF generated by this technique are reported to be (syntactically) smaller than the results of FOCI. In comparison, the interpolation systems discussed in Section 3.5.2 support a strict super-set of EUF. Fuchs' work has recently been extended to combined theories [GKT09]. Section 3.5.1 presents an inference system for a single theory, which can be integrated in Fuchs' framework. An interpolating decision procedure for the theory of unit-to-variable-per-inequality ($\mathcal{UTVPI}\star$), a logic with atoms of the form $(0 \leq ax_1+bx_2+k)$ over $\mathbb{Z}$, is presented in [CGS09]. Jain et al. [JCG09] presents an interpolating decision procedure for linear modular equations, but does not support uninterpreted functions. Brillout et al. [BKRW10] presents an interpolation procedure (based on the sequent calculus) for quantifier-free Presburger arithmetic. Interpolation techniques for modular and Presburger arithmetic are most relevant to our work, since, unlike linear arithmetic, these formalisms allow a faithful representation of bit-vector formulae. Both approaches, however, lack support for bit-wise operations, which are at least partially supported by our inference system.

Our algorithm can also be implemented in a Nelson-Oppen or SMT framework, and interpolants can be generated using the mechanisms presented in [YM05] or [GKT09, BCF+06]. An overview over interpolation techniques based on SMT solvers is provided in [CGS10]. Moreover, [CGS10, Section 3.3] suggests a heuristic to strengthen interpolants for linear arithmetic. Essentially, the approach is based on performing the elimination of existential quantifiers on the partial interpolants contributed by A as dictated by a given proof while setting the partial interpolants contributed by B to true. Intuitively, this re-

sults in an interpolant which is structurally closer to the strongest interpolant specified in Lemma 3.2.1.

Esparza, Kiefer, and Schwoon [EKS06] use binary decision diagrams (BDDs) [Bry86] to generate propositional interpolants. Since the underlying data structure allows the efficient elimination of quantifiers, this approach enables the construction of the strongest and weakest interpolant (see Lemma 3.2.1). In many applications, however, BDDs have been superseded by SAT solvers for reasons of scalability.

[RSS07] proposes to use a constraint solver for linear arithmetic to find coefficients for formulae that enable the elimination of symbols local to a partition. Unlike all other techniques discussed in this section, this approach does not extract interpolants from refutation proofs.

Chapter 4

Counterexamples and Refinement with Loops

Iterative constructs are an integral feature of most programming languages. Cyclic control flow structures and loop invariants are intrinsically tied together, and iterative programming constructs often disclose crucial information about inductive invariants. Chapters 2 and 3, however, neglect the existence of repetitive constructs (despite the fact that Hoare as well as Dijkstra provide mechanisms to reason about loops [Hoa69, Dij75]). This chapter addresses the problems arising from this omission and presents techniques which exploit the structural information provided by iterative constructs.

CEGAR (described in Section 2.5.4) in combination with the refinement techniques presented in Section 2.5.3 guarantees the elimination of the spurious counterexample triggering the refinement step. In the presence of loops iterating over induction variables this restrained refinement strategy may have the adversary effect of enumerating the loop iterations in the abstract domain. A refinement strategy which is unaware of the similarity between two paths which traverse the same loop a different number of times but are otherwise identical treats these counterexamples as entirely separate instances. In the worst case, CEGAR eliminates one such spurious counterexample at a time, thus delaying the detection of counterexamples and inductive invariants. In such a case, the refinement of counterexamples which traverse “deep” loops turns out to be the Achilles’ heel of CEGAR.

The problem affects the detection of counterexamples as well as the verification of programs.

- In order to detect a counterexample that traverses a program loop repeatedly, CEGAR may require one refinement cycle for each iteration of the loop. Accordingly, the abstract counterexample is gradually extended until it reaches the required length. The motivating example in Section 1.3.2 illustrates this behaviour.
- Similarly, if the program is safe, the iterative refinement strategy may result in the construction of an exact model of the induction variable. Accordingly, the loop invariant is approximated from below. Example 2.5.6 demonstrates that this approach may generate diverging sequence of refinement predicates which are insufficient to derive an inductive invariant.

The issue of detecting counterexamples with loops is discussed in Section 4.1. Section 4.2 focuses on refinement in the presence of loops. These techniques enable the verification and falsification of programs which can not be dealt with using the "uninformed" CEGAR approach discussed in Chapter 2. Instances of such programs are presented in Section 4.3. Finally, Section 4.4 provides a characterisation of the class of programs for which our loop detection approach succeeds.

Contribution. This chapter presents techniques to detect potential loops and their respective induction variables. This information is then exploited to either quickly detect counterexamples which would otherwise only be discovered after a large number of refinement steps, or to eliminate a family of spurious counterexamples in a single refinement step.

The content of this chapter has previously been published in [KW10]. In this chapter, we present an adapted and extended version of this paper.[1] The journal paper [KW10] in turn builds upon the work presented in [KW06]. Both publications are joint work with Dr. Daniel Kroening, who provided a prototypical implementation of the counterexample

[1]With kind permission from Springer Science+Business Media: Formal Aspects of Computing, Verification and falsification of programs with loops using predicate abstraction, volume 22, issue 2, 2010, pages 105–128, Daniel Kroening and Georg Weissenbacher, DOI 10.1007/s00165-009-0110-2, BCS © 2009

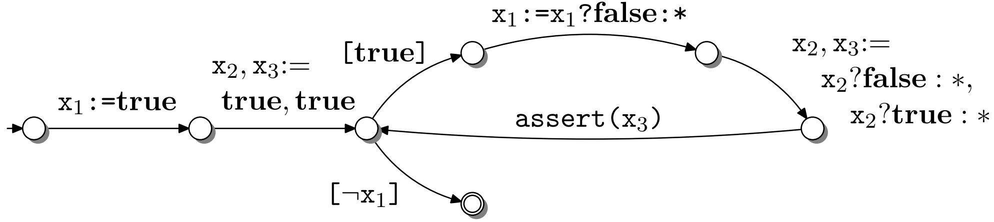


Figure 4.1: A refined version of the transition system in Figure 2.5(b). The variables $\mathtt{x}_1$ and $\mathtt{x}_2$ represent the predicates ($\mathtt{i} = 0$) and ($\mathtt{j} = 0$), respectively. The variable $\mathtt{x}_3$ corresponds to the predicate ($\mathtt{j} < 10$).

detection algorithm in [KW06]. This work is extended and generalised by the author of this dissertation in [KW10].

4.1 Counterexamples with Loops

The refinement techniques presented in Section 2.5.3 fail to take advantage of readily available information about the control flow structure of programs. The refinement algorithms do not distinguish between spurious counterexamples which traverse loops repeatedly and loop-free counterexamples. We illustrate in Example 4.1.1 in Section 4.1.1 that this uninformed strategy may lead to an enumeration of loop iterations. We emphasise that this problem equally applies to interpolation-based as well as predicate abstraction-based approximations, but restrict our presentation to the latter.

4.1.1 How Predicate Abstraction Handles Loops

In order to detect a counterexample that contains several loop iterations, predicate abstraction may require at least one predicate for each iteration of the loop.[2] This is illustrated by the following example.

Example 4.1.1. *Consider the abstract transition system in Figure 4.1, which rules out the spurious counterexample*

$$\mathtt{x}_1 := \textit{true};\ \mathtt{x}_2 := \textit{true};\ [\textit{true}];\ \mathtt{x}_1 := \mathtt{x}_1?\textit{false} : *;\ \mathtt{x}_2 := \mathtt{x}_2?\textit{false} : *$$

[2]Technically, $\lceil \log_2(n) \rceil$ predicates are sufficient to enforce n iterations through a loop in the abstract Boolean program. However, we are not aware of any predicate abstraction-based tool that generates a binary encoding of the loop counter.

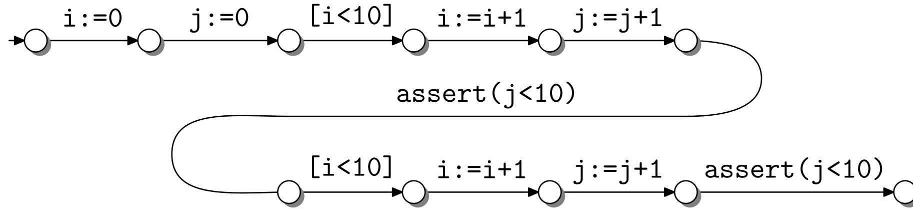


Figure 4.2: A concrete path with two loop iterations

present in Figure 2.5 in Example 2.5.5. This is achieved by adding the predicate $(\mathtt{j} < 10)$, *which we obtain by means of computing the weakest precondition for the concrete counterexample trace. Unfortunately, the refined transition system does not exclude the path that traverses the cycle in the control flow automaton* twice. *Figure 4.2 shows the corresponding concrete path reaching the assertion.*

Again, adding a refinement predicate (namely $\mathtt{j} + 1 < 10$, *which we can derive from the path in Figure 4.2) to* $\mathcal{P}$ *eliminates the spurious counterexample that iterates the loop twice. This predicate, however, fails to eliminate the spurious counterexample that executes the loop three times. In order to find the only unsafe path, which iterates the loop ten times, it is necessary to add all predicates from* $(\mathtt{j} + 1 < 10)$ *up to* $(\mathtt{j} + 9 < 10)$. *To achieve this, at least ten refinement steps are required. The abstraction refinement scheme uses the refinement predicates to encode a "one-hot" counter modelling the concrete induction variable* j.

A large number of refinement predicates may make computational effort required to verify the abstract model prohibitively large. In [BKW07b] we observe run-times of more than one hour for Boolean program instances with 60 predicates. The effort increases exponentially with each iteration of the CEGAR process (see [KW06] and Section 4.5, for instance). The number of predicates in Boolean programs generated by predicate abstraction in the presence of deep loops may easily exceed this limit. For successful verification attempts, the number of predicates is typically much lower. Ball et al. [BBC+06] reports that the average number of predicates required to verify Windows device drivers is eight.

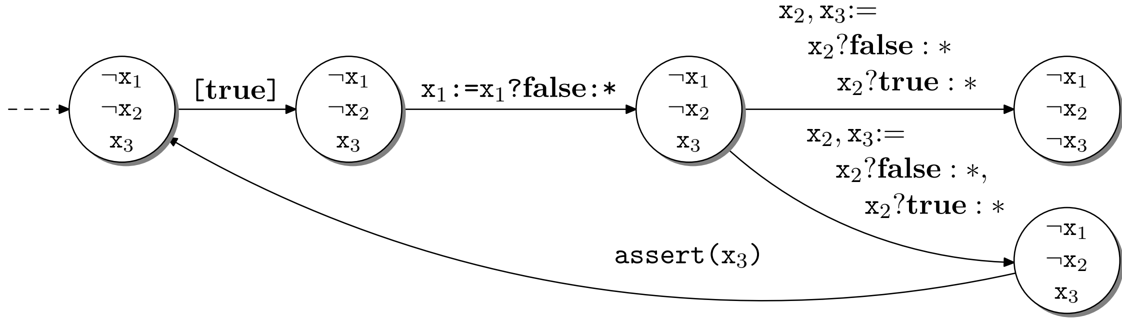


Figure 4.3: Detecting potential loops in abstract paths

4.1.2 Detecting Loops in Abstract Counterexamples

By construction, the verification process (discussed in Section 2.5.1) or the model checking algorithm reports only one path reaching the assertion. The abstract transition system in Figure 4.1 contains not only a single counterexample, but a *family* of similar counterexamples:

Example 4.1.2. *Consider the path π in Figure 4.2. We examine the abstract states of the corresponding abstract path, which we obtain by mapping π to a path of the abstract transition system in Figure 4.1. Starting with an arbitrary state, we reach an abstract state* $x_1 = \textit{true}$, $x_2 = \textit{true}$, *and* $x_3 = \textit{true}$ *after two transitions. The first iteration of the loop changes this state to* $x_1 = \textit{false}$, $x_2 = \textit{false}$, *and* $x_3 = \textit{true}$ *(the first state in Figure 4.3). Once we reach the transition* $x_2, x_3 := x_2?\textit{false} : *, x_2?\textit{true} : *$*, the non-deterministic transition function allows us to make a choice: Either we change* x_3 *to* $\textit{false}$*, which results in a violation of the assertion in the subsequent transition, or we do not change* x_3 *and iterate the loop once more (see Figure 4.3). Alternatively, the program may terminate without violating the assertion. The transition system in Figure 4.1 allows to iterate the loop arbitrarily often before the assertion is finally violated.*

This suggests that there is a potential loop in the original program (as indicated by the repetition signs ‖: and :‖ in Figure 4.4). The model checking tool, though, reports only a finite path π (namely the *shortest* path potentially violating the assertion), but does not provide information on potential loops that are traversed by this path.

The missing information can be added using a post-processing step: The algorithm

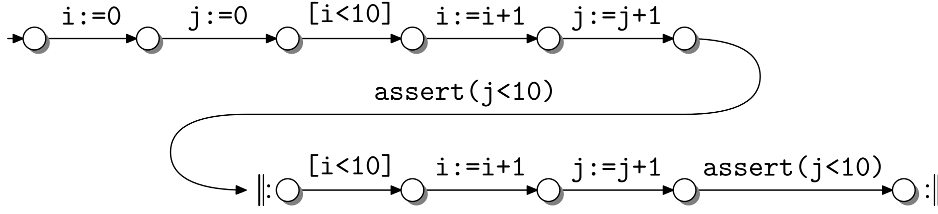


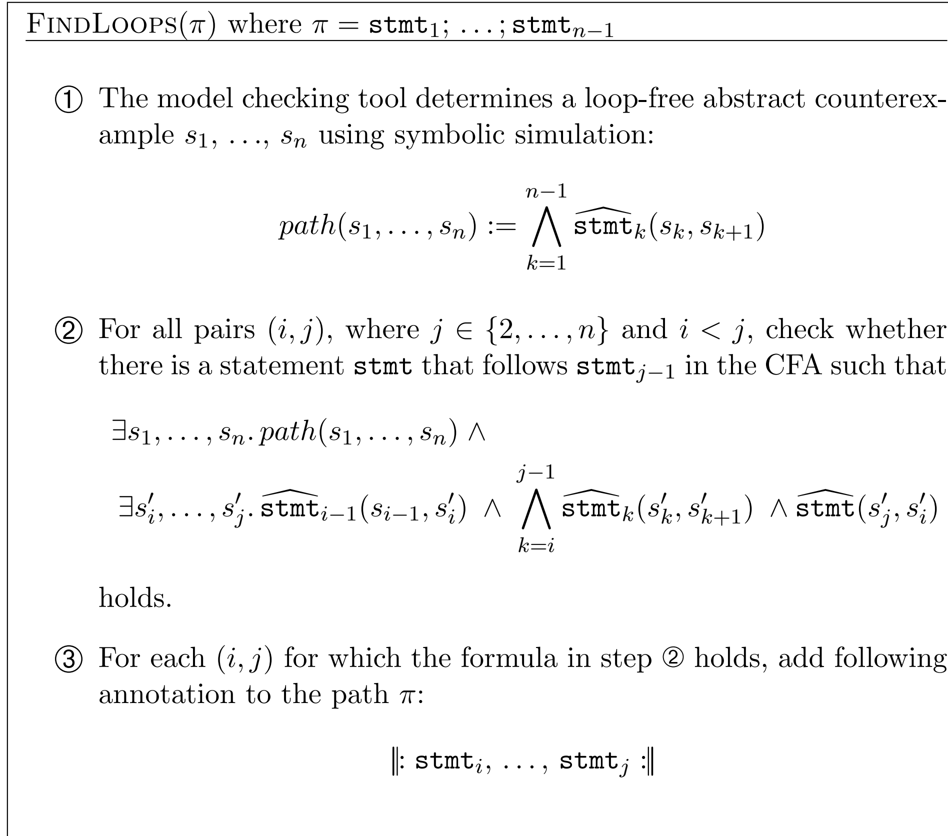
Figure 4.4: The path from Figure 4.2 annotated with loop information

presented in Figure 4.5 searches for loops in abstract counterexamples. We use the relational notation for statements introduced in Section 2.4.2. A loop has to contain a back-edge that allows us to jump back to an earlier location in the path π. Using the techniques discussed in Section 3.3, we construct a propositional formula (in step ②) that enables us to efficiently search for back-edges $\widehat{\texttt{stmt}}$ in the abstract transition system. Intuitively, step ② corresponds to a model checking run that checks for each location i in π whether there is a path that leads back to i visiting only locations occurring in π. An example for such a back-edge is the transition labelled `assert(x₃)` in Figure 4.3 (as explained in Example 4.1.2).

This idea is formalised in Figure 4.5. The conjunction of abstract transition relations in step ① and ② can be encoded as a propositional formula, for which satisfiability (SAT) is usually efficiently decidable [ES04]. The predicate $path(s_1, \ldots, s_n)$ is a symbolic representation of the abstract counterexample π. The algorithm tries to find back-edges in the *abstract* transition system for each sub-path $\texttt{stmt}_i;\ldots,\texttt{stmt}_j$ of π, resulting in a quadratic number of SAT-instances. Our experiments show that the overhead for detecting loops is negligible compared to the time the model checking tool spends searching for counterexamples. Note that the algorithm is also able to detect nested loop structures. In the case that the verification tool is not based on predicate abstraction (see [McM06], for instance), the detection of abstract loops boils down to checking whether the counterexample visits a loop head repeatedly.

4.1.3 Checking the Safety of Counterexamples with Loops

The existence one or more loops in the abstract counterexample does not imply that there is a corresponding path that violates the assertion at the end of the counterexample π. The question whether a counterexample with loops is safe is undecidable in general. It is, how-



FINDLOOPS(π) where $\pi = \texttt{stmt}_1; \ldots; \texttt{stmt}_{n-1}$

① The model checking tool determines a loop-free abstract counterexample $s_1, \ldots, s_n$ using symbolic simulation:

$$path(s_1, \ldots, s_n) := \bigwedge_{k=1}^{n-1} \widehat{\texttt{stmt}}_k(s_k, s_{k+1})$$

② For all pairs (i, j), where $j \in \{2, \ldots, n\}$ and $i < j$, check whether there is a statement $\texttt{stmt}$ that follows $\texttt{stmt}_{j-1}$ in the CFA such that

$$\exists s_1, \ldots, s_n . path(s_1, \ldots, s_n) \wedge$$
$$\exists s'_i, \ldots, s'_j . \widehat{\texttt{stmt}}_{i-1}(s_{i-1}, s'_i) \wedge \bigwedge_{k=i}^{j-1} \widehat{\texttt{stmt}}_k(s'_k, s'_{k+1}) \wedge \widehat{\texttt{stmt}}(s'_j, s'_i)$$

holds.

③ For each (i, j) for which the formula in step ② holds, add following annotation to the path π:

$$\|\!:\ \texttt{stmt}_i, \ldots, \texttt{stmt}_j\ :\!\|$$

Figure 4.5: Algorithm to detect loops in abstract counterexamples

ever, possible to obtain a loop-free instance of the annotated counterexample by *unrolling* the loops a certain number of times. Using forward symbolic simulation (which is equivalent to computing the strongest post-condition of the path), we can then determine whether this instance violates the assertion or not. Consider, for instance, the counterexample π_9 obtained by unrolling the loop in Figure 4.2 nine times. Let π be the prefix of π_9 that does not contain the last assertion of π_9. Then, we can show that $\neg(sp(\pi, \mathsf{true}) \Rightarrow (\mathtt{j} \geq 10))$ is satisfiable (or even that $\{\mathsf{true}\}\ \pi\ \{\mathtt{j} \geq 10\}$ holds), i.e., that π constitutes a concrete counterexample and that the program is therefore not safe.

In general, we do not know how many times we need to unroll the loops to obtain an unsafe path (it might even be that there is no such path). Therefore, we use a heuristic to find promising candidates.

The first step is to convert the counterexample π into static single assignment form (SSA) [CFR+91]. The SSA form is a representation in which each variable of a program

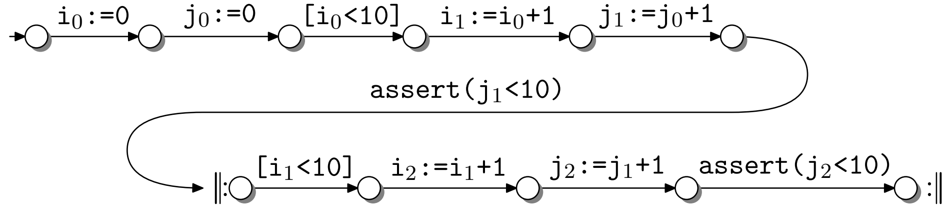


Figure 4.6: The path from Figure 4.4 converted to Single Static Assignment form

is assigned exactly once (see page 29). For this purpose, we replace each existing variable by an indexed version of this variable (see Figure 4.6, for example). Whenever we reach an assignment, the version number is increased by one.[3]

As we unroll the loop, a pattern emerges: In each iteration, the variables $\mathtt{i}_n$ and $\mathtt{j}_n$ depend on $\mathtt{i}_{n-1}$ and $\mathtt{j}_{n-1}$, respectively (see Figure 4.7). Since the counterexample is in SSA form, it is easy to identify variants of the loop by means of a simple syntactic analysis. We obtain a *recurrence equation* for each variable that is changed in the loop:

$$\mathtt{i}_0 = 0\,, \quad \mathtt{i}_1 = \mathtt{i}_0 + 1 \qquad \mathtt{i}_n = \mathtt{i}_{n-1} + 1 \tag{4.1}$$

$$\mathtt{j}_0 = 0\,, \quad \mathtt{j}_1 = \mathtt{j}_0 + 1 \qquad \mathtt{j}_n = \mathtt{j}_{n-1} + 1 \tag{4.2}$$

We proceed by computing the *closed form* of these recurrence equations. Computing the closed form of an arbitrary recurrence equation is a non-trivial problem. In fact, in some cases such a closed form may not even exist [vEBG04]. In many real-world programs, however, the recurrence equations that occur in a loop are relatively simple. In our implementation, we consider only recurrence equations of the form

$$\mathtt{i}_0 = \alpha, \quad \mathtt{i}_n = \mathtt{i}_{n-1} + \beta + \gamma \cdot n$$

(where $n > 0$ and α, β, and γ are numeric constants or loop-invariant symbolic expressions and $\mathtt{i}$ is the variant). According to [GKP89], the corresponding closed form is

$$\mathtt{i}_n = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}\,.$$

[3] The result of this process is similar to what we obtain by computing the strongest post-condition and eliminating the existential quantifiers using Skolemisation.

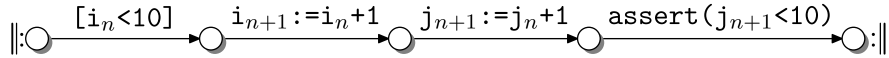


Figure 4.7: A pattern derived by unrolling the loop in Figure 4.6

It follows that the closed form of the recurrence equations (4.1) and (4.2) is $\texttt{i}_n = n$ and $\texttt{j}_n = n$, respectively. This translates to $\texttt{i}_{n+1} = \texttt{i}_1 + n$ and $\texttt{j}_{n+1} = \texttt{j}_1 + n$ in our example (see Figures 4.6 and 4.7). The variable n corresponds to the number of loop iterations. If we replace the right-hand sides of the assignments in the loop in Figure 4.6 by their corresponding closed forms, we obtain a counterexample that is *parametrised* with the number of loop iterations n. The instructions of the parametrised counterexample constrain the indexed variables that occur in the path as well as the variable n. According to Lemma 2.2.1, it is possible to represent this path as an existentially quantified formula. Accordingly, Figure 4.8 shows the parametrised loop and the formula $F_\pi(n)$ derived from the counterexample. Note that the condition contributed by the last assertion is negated (cf. observation 3 on page 37). Using a constraint solver, we try to compute the smallest value of n that occurs in a satisfying assignment of the formula $F_\pi(n)$.

The smallest value of n that is part of a satisfying assignment to the formula in Figure 4.8 is 9. This value is an educated guess for the number of iterations necessary to violate the assertion. We unwind the loop of the counterexample according to the pattern in Figure 4.7 such that the last assignment is $\texttt{j}_{10} := 10$ and the last assertion is $\texttt{assert}(\texttt{j}_{10} < \texttt{10})$. As explained above, this path constitutes a concrete counterexample showing that the program is not safe.

Figure 4.9 shows the algorithm we use to compute candidates for the number of iterations of the loops in a counterexample. Note that the algorithm is able to handle more than one loop, and even nested loops. If the algorithm encounters a recurrence equation for which it fails to compute its closed form,[4] then the corresponding variable is assigned non-deterministically, i.e., the assignment does not contribute a constraint to the formula. Even though this may result in wrong guesses for the number of iterations, the soundness of the

[4]While it would certainly be feasible to support a larger class of recurrence equations (see for instance [vEBG04]), it turns out that our approach is sufficient to cover the most common cases like linear counters. We do not need to support cases in which the loop counter increases exponentially: These cases can be handled efficiently by traditional unwinding if the bounded range of the program variables is taken into account.

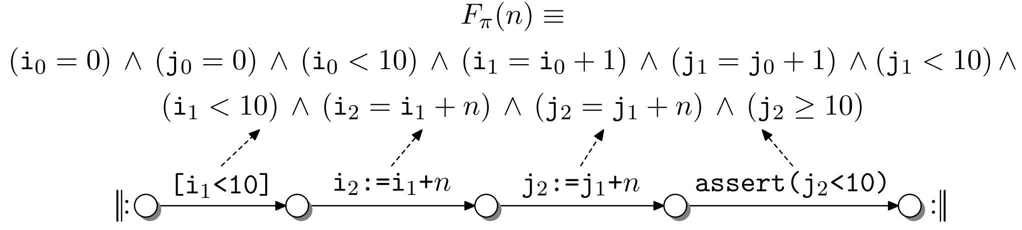


Figure 4.8: Constraints derived from the parametrised loop

approach is guaranteed by checking the safety of the unwound counterexample by means of forward symbolic simulation.

In general, the parametrised formula is not necessarily satisfiable. If, however, the path π constitutes a counterexample, then $F_\pi(n)$ is satisfiable. This follows from the fact that step ② in Figure 4.9 constructs a correct upper approximation of the strongest postcondition.

Lemma 4.1.1. *Let π be the SSA-form of a path, i.e., each assignment to a variable $\mathtt{i}_n$ ($n \geq 1$) is preceded by an assignment to $\mathtt{i}_{n-1}$. Furthermore, let α, β, and γ be numeric constants or invariant symbolic expressions in π, and let $f(\mathtt{i}_n)$ denote an arbitrary (but fixed) function over $\mathtt{i}_n$, α, β, and γ. Given a sequence of (not necessarily consecutive) assignment statements $\mathtt{i}_{n-1}{:=}\alpha$ and $\mathtt{i}_n{:=}f(\mathtt{i}_{n-1})$ (where $n \in \{1..m\}$ for some $m \geq 0$) which occur in π, let π' be a path obtained by replacing all assignments $\mathtt{i}_n{:=}f(\mathtt{i}_{n-1})$ in π with*

$$\begin{cases} \mathtt{i}_n := \alpha + \beta \cdot n + \gamma \frac{n\cdot(n+1)}{2} & \text{if } f(\mathtt{i}_{n-1}) = \mathtt{i}_{n-1} + \beta + \gamma \cdot n \\ \mathtt{i}_n := * & \text{otherwise.} \end{cases}$$

Then it holds that $sp(\pi, \textsf{true}) \Rightarrow sp(\pi', \textsf{true})$.

Proof. By induction on m. Let π_m be a prefix of π such that all for assignments $\mathtt{i}_n{:=}f(\mathtt{i}_{n-1})$ in π_m it holds that $n \leq m$ (and similarly for π'_m and π'). If $m = 0$, then $\pi_0 = \pi'_0$, which constitutes the base case. The induction hypothesis is that $sp(\pi_m, \textsf{true}) \Rightarrow sp(\pi'_m, \textsf{true})$. W.l.o.g., let π_{m+1} be π_m; `stmt`. We perform a case split for `stmt`.

- If `stmt` is not an assignment of the form $\mathtt{i}_{m+1}{:=}f(\mathtt{i}_m)$, then it follows immediately from the monotonicity of the strongest postcondition and the induction hypothesis that $sp(\pi_m; \mathtt{stmt}, \textsf{true}) \Rightarrow sp(\pi'_m; \mathtt{stmt}, \textsf{true})$

GUESSITERATIONS(π), where π is annotated with loops

① Transform π into SSA.

② For each loop in π (as reported by FINDLOOPS):

❶ Generate a recurrence equation for each variable that is updated in the loop.

❷ Introduce a fresh variable n for the loop and try to calculate the closed forms for the recurrence equations. If unable to compute a closed form, leave the corresponding variable unconstrained (i.e., assign it non-deterministically).

$$\left.\begin{array}{lcl} \mathtt{i}_0 & = & \alpha \\ \mathtt{i}_n & = & \mathtt{i}_{n-1} + \beta + \gamma \cdot n \end{array}\right\} \longrightarrow \mathtt{i}_n = \alpha + \beta \cdot n + \gamma \frac{n \cdot (n+1)}{2}$$

$$\text{otherwise} \longrightarrow \mathtt{i}_n = *$$

❸ Substitute the right-hand sides of the assignments in π with the corresponding closed forms.

③ Generate the constraints for the parametrised path. Search for a satisfying assignment for the corresponding formula that minimises the parameters $\{n_1, n_2, \ldots\}$ introduced in step ②. If such an assignment exists, return the parameter values. Otherwise, return "unsatisfiable".

Figure 4.9: Computing the number of iterations for a counterexample with loops

- If `stmt` is an assignment of the form $\mathtt{i}_{m+1} := \mathtt{i}_m + \beta + \gamma \cdot (m+1)$ then

$$sp(\pi_m; \mathtt{i}_{m+1} := \mathtt{i}_m + \beta + \gamma \cdot (m+1), \mathsf{true}) \Rightarrow$$
$$sp(\pi'_m; \mathtt{i}_{m+1} := \alpha + \beta \cdot (m+1) + \gamma \frac{(m+1) \cdot (m+2)}{2}, \mathsf{true})$$

follows from the monotonicity of the strongest postcondition and the equivalence of recurrence equations and their closed form [GKP89]. The implication still holds if we replace $m+1$ with a fresh variable.

- Otherwise, since $\mathtt{i}_{m+1}$ does not occur in π_m or π'_m, it holds that

$$\begin{aligned} sp(\pi_m;\ \mathtt{i}_{m+1}{:=}f(\mathtt{i}_m), \mathsf{true}) &\Rightarrow \exists v \,.\, \mathtt{i}_{m+1} = v \wedge sp(\pi_m, \mathsf{true}) \\ &\Rightarrow \exists v \,.\, \mathtt{i}_{m+1} = v \wedge sp(\pi'_m, \mathsf{true}) \\ &= sp(\pi'_m;\ \mathtt{i}_{m+1}{:=}\ *, \mathsf{true})\,, \end{aligned}$$

  where v is a fresh variable not occurring in $sp(\pi_m, \mathsf{true})$ and $sp(\pi'_m, \mathsf{true})$.

□

On the other hand, the satisfiability of the formula $F_\pi(n)$ does not imply that unwinding the loop gives us a counterexample that violates the assertion. In Section 4.2, we discuss how to refine the abstract transition system if our heuristic fails.

4.2 Refinement in the Presence of Loops

The heuristic presented in Section 4.1.3 may fail to determine the number of iterations necessary to obtain an unsafe path. We distinguish two causes of failure: Either there is no valuation to the parameters $\{n_1, n_2, \ldots\}$ that satisfies the constraints of the path, or the heuristic suggests values for the parameters that result in an unwound counterexample that does not violate the assertion. We discuss the former cause in Section 4.2.1, and the latter cause in Section 4.2.2.

4.2.1 Refinement Using Closed Recurrence Equations

Figure 4.10 shows a safe program. The traditional refinement approach described in Section 2.5.3 may require ten refinement steps to introduce all predicates necessary to show the safety of the program (unless the predicate transformer used for refinement yields the interpolant $\mathtt{i} = \mathtt{j}$). After two abstraction refinement cycles, the FINDLOOPS algorithm (Figure 4.5) detects a loop in the abstract transition relation. The suffix of the corresponding annotated concrete path is shown in Figure 4.11.

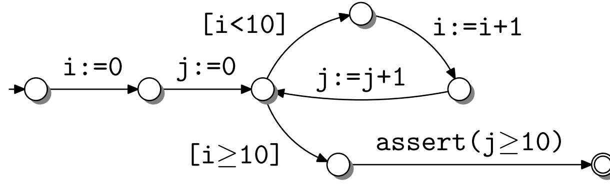


Figure 4.10: A safe program

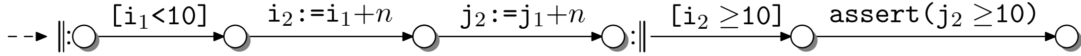


Figure 4.11: A loop detected in the abstraction of the program in Figure 4.10

GuessIterations in Figure 4.9 fails to compute a valuation for the parameter n, since the formula

$$\ldots \wedge (\mathtt{i}_1 < 10) \wedge (\mathtt{i}_2 = n) \wedge (\mathtt{j}_2 = n) \wedge (\mathtt{i}_2 \geq 10) \wedge (\mathtt{j}_2 < 10) \tag{4.3}$$

is unsatisfiable. The formula derived from the parametrised path may have more satisfying assignments than the formula corresponding to the loop-free path. This stems from the potentially introduced non-determinism and from the fact that it encodes an arbitrary number of iterations of the loop (cf. Lemma 4.1.1). Therefore, its unsatisfiability implies the infeasibility of the original counterexample. Therefore, it is of course possible to fall back to the traditional refinement approach (using predicate transformers and the limited set of rules in Figure 2.1) for the counterexample *without loops* to refine the abstract transition relation. This approach, however, does not exploit the knowledge we have about the loop in the abstract program. Instead, we take advantage of this information by using a Hoare logic rule that allows us to reason about loops: Figure 4.12 shows the rule for `while`-loops. It states that, given P is an invariant of the loop body π_1, the execution of the entire loop also maintains this invariant. This rule is not as easy to apply as the rules in Figure 2.1: There is no general technique to automatically infer a loop invariant P that is strong enough to show the safety of a path with loops.

$$\frac{\{P \wedge Q\}\ \pi_1\ \{P\}}{\{P\}\ \|\!:\ [Q]\,;\,\pi_1\ :\!\|\ [\neg Q]\ \{\neg Q \wedge P\}}\ \text{loop}$$

Figure 4.12: Hoare logic rule for loops

In our example, we already have a sufficiently strong loop invariant at hand: The conjunction of the closed forms of the recurrence equations $(\mathtt{i} = 1+n)$ and $(\mathtt{j} = 1+n)$ implies that $(\mathtt{i} = \mathtt{j})$ is an invariant of the loop in Figure 4.11. This predicate is sufficient to show that $(\mathtt{i} \geq 10) \wedge (\mathtt{j} < 10)$ can not hold. Therefore, we can apply the rule in Figure 4.12 to show the safety of the program and of the path in Figure 4.11. By applying the loop rule to the loop in Figure 4.10, we infer

$$\{(\mathtt{i} = \mathtt{j})\} \;\|\!\!:\; [\mathtt{i} < 10]; \mathtt{i} := \mathtt{i}+1; \mathtt{j} := \mathtt{j}+1 \;:\!\!\|\; [\mathtt{i} \geq 10] \; \{(\mathtt{i} \geq 10) \wedge (\mathtt{i} = \mathtt{j})\}\,. \tag{4.4}$$

The prefix $\mathtt{i} := \mathtt{0}$; $\mathtt{j} := \mathtt{0}$ establishes the precondition. Note that $(\mathtt{i} = \mathtt{j})$ is an inductive invariant for the loop. Since the postcondition of (4.4) implies $(\mathtt{j} \geq 10)$, the assertion cannot be violated.

One might argue that the interpolation techniques presented in Chapter 3 already enable us to find the inductive invariant $(\mathtt{i} = \mathtt{j})$. However, the closed forms of the recurrence equations are *always* loop invariants for their corresponding parametrised counterexamples. Adding these loop invariants to the set of predicates $\mathcal{P}$ can result in a significantly smaller number of refinement iterations. Note that these predicates refer to the parameters $\{n_1, n_2, \ldots\}$ introduced by GuessIterations (see Figure 4.9). The interpolation algorithms in Chapter 3 enable us to eliminate these parameters from the predicates (by means of quantifier elimination). In general, however, predicates generated using the weakest precondition or strongest postcondition may contain parameters. Therefore, instead of trying to eliminate $\{n_1, n_2, \ldots\}$ from the predicates, we instrument the original program with corresponding induction variables (see Figure 4.13). For each loop, we introduce an *induction variable* n_i, which is initialised before the loop is entered, and increased at the end of the loop body. This modification has no impact on the safety of the current counterexample or the program. The advantage of this approach is that it is not necessary to modify the abstraction algorithm (see Section 2.4.2): In order to make sure that the information about the loop invariant is preserved, we add *three* versions of the recurrence predicate P to $\mathcal{P}$, namely $P[n/0]$, P, and $P[n/n+1]$. Then, the resulting abstraction is strong enough to show that P is preserved when the loop is traversed along the path of the

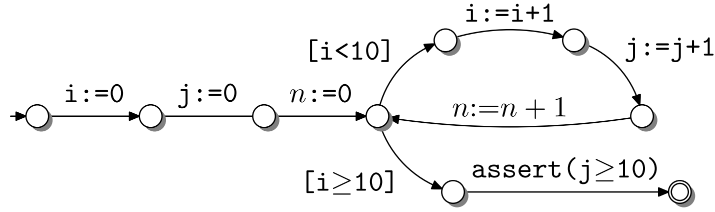


Figure 4.13: The program of Figure 4.10 augmented with an induction variable n

spurious counterexample:

1. First, $\{P[n/0]\}$ n:= 0 $\{P\}$ establishes P upon entrance to the loop.

2. Let `stmt` be the statement that modifies the induction variable. Then the Hoare triple $\{P\}$ `stmt` $\{P[n/n+1]\}$ holds.

3. Finally, by the assignment rule, $\{P[n/n+1]\}$ $n{:=}n+1$ $\{P\}$ holds upon exit from the loop.

Intuitively, we avoid the universal quantification over n by an induction over the parameter n. In our example in Figure 4.13, the predicates ($\mathtt{i} = 0$), ($\mathtt{i} = n$), ($\mathtt{i} = n{+}1$), ($\mathtt{j} = 0$), ($\mathtt{j} = n$), and ($\mathtt{j} = n+1$) are sufficient to prove the loop invariant ($\mathtt{i} = n$) $\wedge$ ($\mathtt{j} = n$). In combination with the predicates ($\mathtt{i} \geq 10$) and ($\mathtt{j} \geq 10$), this invariant is strong enough to show the safety of all unwindings of the counterexample in Figure 4.11.

The loop invariant given by the conjunction of the recurrence predicates is not always strong enough to eliminate the spurious counterexample. The reason is that the approach presented above ignores all statements in the loop body except the ones that contribute a recurrence equation. We observe that the original counterexample is also infeasible, since the parametrised path is more general than the original path. Therefore, adding the predicates that we can extract from the non-parametrised path is sufficient to eliminate the original counterexample. In addition, we add the recurrence predicates, hoping that they eliminate other potential counterexamples that contain more than one loop iteration from the model.

Predicate abstraction is able to establish disjunctive invariants that can be expressed in terms of the predicates [BMMR01]. Therefore, our technique is also able to establish the safety of programs with disjunctive loop invariants. In Section 4.3, we will give an example for such a program.

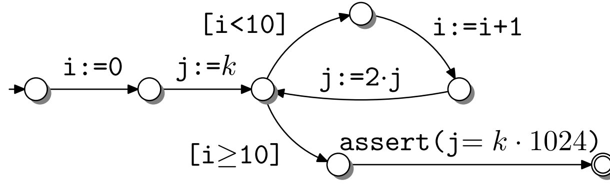


Figure 4.14: A safe program

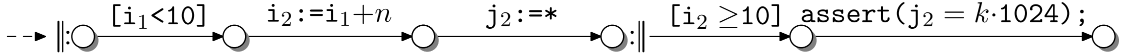


Figure 4.15: A parametrised path with a loop for the program in Figure 4.14

4.2.2 Refinement Using Unwound Spurious Counterexamples

The scheme we use to solve recurrence equations matches only the cases specified in step ②.❷ of Figure 4.9. It fails to solve recurrences as simple as $\mathtt{j}_n = 2{\cdot}\mathtt{j}_{n-1}$. Therefore, the approach described in Section 4.1.3 yields the parametrised path in Figure 4.15 for the program in Figure 4.14. The resulting formula does not constrain the variable $\mathtt{j}_2$:

$$\ldots \wedge (\mathtt{i}_1 < 10) \wedge (\mathtt{i}_2 = n) \wedge (\mathtt{i}_2 \geq 10) \wedge (\mathtt{j}_2 \neq \mathtt{k} \cdot 1024) \tag{4.5}$$

GuessIterations determines that 10 is the smallest value for n such that Formula (4.5) is satisfiable. The corresponding unwound path, however, is safe. Even though the predicate $(\mathtt{i} = n)$ is a loop invariant, it is not strong enough to show the safety of the program.

In that case, we fall back on the traditional refinement approach. To eliminate all spurious counterexamples represented by the path with loops, it is necessary to add the refinement predicates from the proof of safety for the unwound counterexample. In our example, the weakest precondition yields the predicates $(\mathtt{j} = 2 \cdot \mathtt{k})$, $(\mathtt{j} = 4 \cdot \mathtt{k})$, $(\mathtt{j} = 8 \cdot \mathtt{k})$, $\ldots$, $(\mathtt{j} = 1024 \cdot \mathtt{k})$ and $(\mathtt{i} < 10)$, $(\mathtt{i} + 1 < 10)$,$\ldots$, $(\mathtt{i} + 9 < 10)$, which are sufficient to show the safety of the path.

An obvious disadvantage of this approach is that it generates a large number of predicates. The traditional refinement technique, however, yields the same set of predicates, but needs at least ten refinement steps, while our technique shows the safety of the program in only three abstraction refinement cycles. In Section 4.5, we present benchmarks for which this eager refinement approach performs better than traditional iterative refinement.

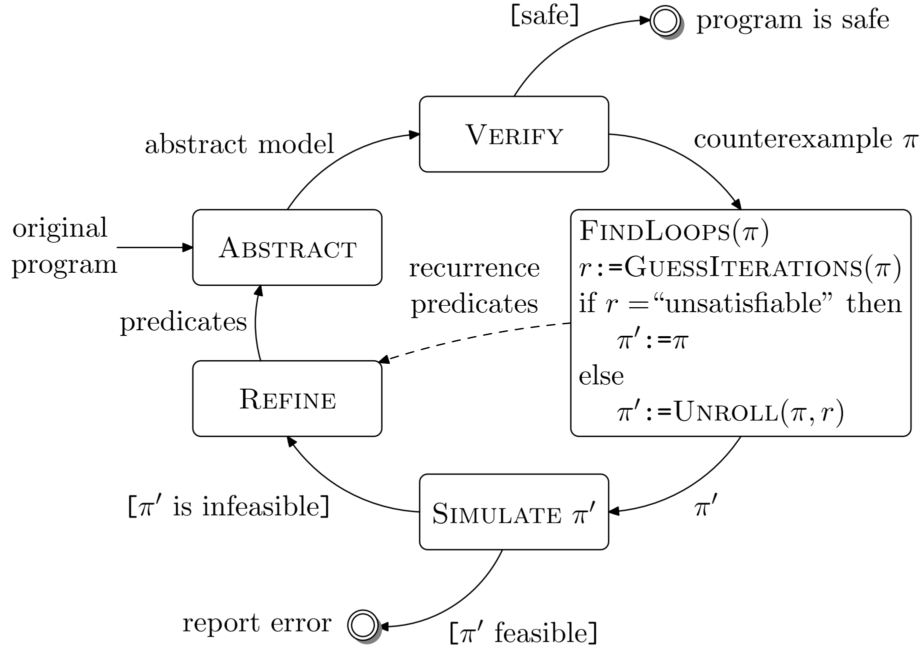


Figure 4.16: Integrating loop detection into the abstraction refinement cycle

Integrating our Techniques into CEGAR Figure 4.16 shows how our loop detection algorithm and the improved refinement technique are integrated into the traditional abstraction refinement cycle (see Figure 2.9 in Section 2.5.4). The analyses presented in Section 4.1 (Figures 4.5 and 4.9) are introduced between the model checking step and the simulation phase. Depending on the result r of the GUESSITERATIONS heuristic, the potential loops in the counterexample π may be unrolled accordingly (indicated by $\text{UNROLL}(\pi, r)$), yielding a loop-free counterexample π'. As explained in Section 4.2, the recurrence predicates are added to the set of predicates P in the refinement step (indicated by the dashed arrow in Figure 4.16). Notably, our approach does not require any major modifications of the original steps of the CEGAR algorithm.

4.3 Examples

Figure 4.17 shows three programs which are slightly more sophisticated than the examples discussed so far. (Recall that we already encountered the example in Figure 4.17(b) in Section 2.5.4, where we demonstrated that CEGAR may fail to terminate.) We discuss how these programs are verified using the approach presented in the previous sections.

Alternating Branches. The program in Figure 4.17(a) is not safe: The assertion can be violated by iterating the loop 40 times. Note that the branches of the conditional statement in the loop body are alternating. Initially, our approach detects a potential loop that repeatedly executes the same branch (for instance, the branch in which i and j are increased). The corresponding parametrised counterexample is infeasible, since x is initialised to 0 in the prefix of the path and never modified when traversing the loop. The predicates ($\mathtt{i} < 20$), ($\mathtt{x} < 20$), and $\neg\mathtt{b}$, which are determined using the traditional refinement approach, eliminate this spurious counterexample and force the execution of both branches of the conditional statement in the correct order. The model checker is now forced to unwind the loop twice and reports a corresponding abstract counterexample. Again, this counterexample contains a potential loop, namely

$$\begin{array}{l} \ldots \|\!: [(\mathtt{i} < 20) \vee (\mathtt{x} < 20)];\ [\mathtt{b}];\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} + 1;\ \mathtt{b} := \neg\mathtt{b}; \\ \quad [(\mathtt{i} < 20) \vee (\mathtt{x} < 20)];\ [\neg\mathtt{b}];\ \mathtt{i} := \mathtt{i} + 1;\ \mathtt{j} := \mathtt{j} + 1;\ \mathtt{b} := \neg\mathtt{b} :\!\|\ \ldots\ . \end{array}$$

The body of this loop is an unwinding of the cycle in the control flow graph in Figure 4.17(a). GuessIterations (see Figure 4.9) yields 20 as a promising candidate for the number of iterations of this loop. Finally, the forward simulation of the corresponding unwound counterexample confirms that it is indeed a feasible path that violates the assertion.

In the setting described above, adding the recurrence predicates $\mathtt{x} = n$, $\mathtt{y} = n$, $\mathtt{i} = n$, and $\mathtt{j} = n$ fails to provide any benefit. If, however, we change the assertion in Figure 4.17(a) to $\mathtt{assert}((\mathtt{j} \geq 20) \wedge (\mathtt{y} \geq 20))$, the resulting loop invariant $(\mathtt{x} = \mathtt{y}) \wedge (\mathtt{i} = \mathtt{j})$ is sufficient to show the safety of the modified program.

Diverging Sequence of Predicates. Figure 2.10 shows a program presented by Jhala and McMillan [JM06]. For this example, the traditional refinement heuristic[5] yields a diverging sequence of predicates insufficient to represent the loop invariant $(\mathtt{i} = \mathtt{j}) \Rightarrow (\mathtt{x} = \mathtt{y})$. Our approach is capable of detecting the loop and inferring the recurrence predicates $\mathtt{x} = \mathtt{i} - n$ and $\mathtt{y} = \mathtt{j} - n$. In combination with the conditions $(\mathtt{i} = \mathtt{j})$ and $(\mathtt{x} = 0)$, the resulting loop

[5] Jhala and McMillan refer to what we call the traditional refinement heuristic as "typical predicate heuristic".

invariant $(\mathtt{x} = \mathtt{i} - n) \wedge (\mathtt{y} = \mathtt{j} - n)$ is sufficient to establish the safety of the program.

Disjunctive and Non-Linear Invariants. The program in Figure 4.17(c) has two interesting aspects: The loop invariant is non-linear; moreover, it is disjunctive. Similar to Figure 4.17(a), the loop contains a conditional statement with two branches. In the program in Figure 4.17(c), however, only one of the two branches can be executed. The non-deterministic assignment `b:=*` determines which branch is selected. Depending on this choice, either the assertion $(\mathtt{i}= 2{\cdot}\mathtt{j})$ or the assertion $(2{\cdot}\mathtt{i}{>}\mathtt{j}^2)$ holds. Therefore, the invariant of the whole loop is the disjunction $(\mathtt{i}= 2{\cdot}\mathtt{j}) \vee (2{\cdot}\mathtt{i}{>}\mathtt{j}^2)$ Our approach detects the two parts of the invariant separately and leaves the task to merge this information into a disjunction to the predicate abstraction framework. Our algorithm detects two alternative loops, namely

$$\Vert\!:\ [\mathtt{j} < \mathtt{10}];\ \mathtt{j} := \mathtt{j} + \mathtt{1};\ [\neg\mathtt{b}];\ \mathtt{i} := \mathtt{i} + \mathtt{2};\ \mathtt{assert}((\mathtt{i} = \mathtt{2} \cdot \mathtt{j}) \vee (\mathtt{2} \cdot \mathtt{i} > \mathtt{j}^2))\ :\!\Vert \quad \text{and}$$

$$\Vert\!:\ [\mathtt{j} < \mathtt{10}];\ \mathtt{j} := \mathtt{j} + \mathtt{1};\ [\mathtt{b}];\ \mathtt{i} := \mathtt{i} + \mathtt{j};\ \mathtt{assert}((\mathtt{i} = \mathtt{2} \cdot \mathtt{j}) \vee (\mathtt{2} \cdot \mathtt{i} > \mathtt{j}^2))\ :\!\Vert\ .$$

The recurrence predicates $(\mathtt{j} = n)$ and $(\mathtt{i} = 2 \cdot n)$ are sufficient to show the safety of the first loop. The latter case is more complicated, since the two assignments are interdependent. We resolve this dependency by processing the recurrence equations in topological order. First, we determine the closed form $\mathtt{j} = n$ for the assignment `j:=j+1`. Using this closed form, we eliminate `j` from the assignment `i:=i+j` and obtain the instruction $\mathtt{i:=i+}n$. For the corresponding recurrence equation, $\mathtt{i}_n\mathtt{:=i}_{n-1}\mathtt{+}n$, we compute the closed form

$$\mathtt{i}_n = \mathtt{i}_0 + \frac{n \cdot (n+1)}{2}$$

(as explained in Figure 4.9). Since $\mathtt{i}_0$ is 0, the resulting recurrence predicate is $\mathtt{i} = \frac{n \cdot (n+1)}{2}$. This predicate, in combination with $\mathtt{j} = n$, implies $2 \cdot \mathtt{i} > \mathtt{j}^2$. Therefore, the recurrence predicates generated by the algorithm proposed in Section 4.2.1 are (in theory) sufficient to prove the safety of the program. In practice, unfortunately, it turns out that most predicate-abstraction-based model checking tools do not support non-linear arithmetic operations at all. Even though our verification tool SatAbs [CKSY05] is able to handle non-linear arith-

metic by converting the operations to propositional formulas, the SAT instances generated during the verification of the program in Figure 4.17(c) turn out to be too complex for the underlying SAT solver.

This negative result, however, must not be mistaken as evidence that our approach does not scale in the presence of non-linear arithmetic in general. For the following counterexample, for instance, our implementation is able to determine 20 as the number of iterations for which the assertion is violated. This takes only two refinement cycles and less than one second:

$$\mathtt{i} = \mathtt{0};\, \mathtt{j} = \mathtt{0};\, \|\!:\ [*];\, \mathtt{j} := \mathtt{j} + \mathtt{1};\, \mathtt{i} := \mathtt{i} + \mathtt{j};\, \mathtt{assert}(\mathtt{i} < \mathtt{210})\ :\!\|$$

(Here, $[*]$ denotes a condition that non-deterministically evaluates to either **true** or **false** in each iteration of the loop.)

4.4 Conditions for Completeness

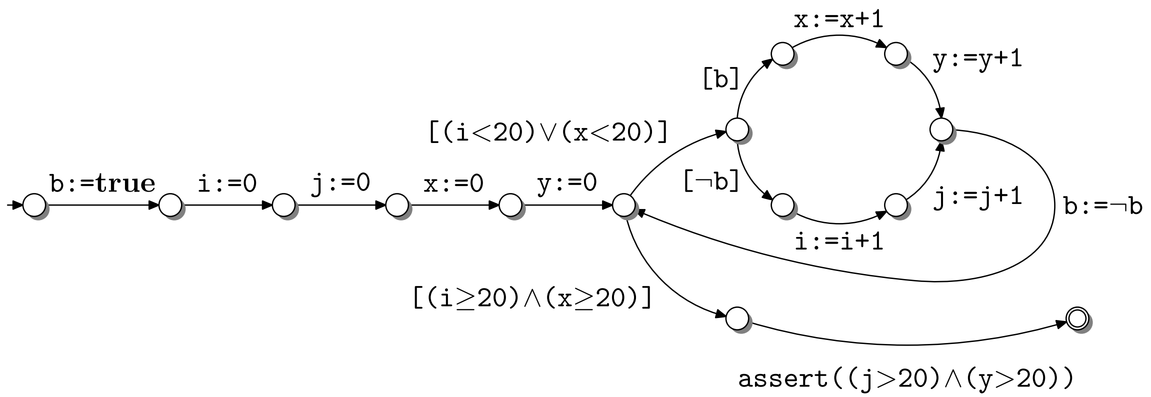
The examples in the previous section raise the question of whether the class of programs for which our approach is complete (i.e., able to prove safety) can be defined rigorously. Even though the traditional CEGAR approach discussed in Section 2.5.4 may succeed to establish an invariant that is sufficiently strong to show the safety (this is the case for the program in Figure 4.10, for instance), it fails to do so in general: As discussed in Section 2.5.4, the traditional refinement algorithm yields a sequence of diverging predicates for the program in Figure 2.10. Our algorithm is able to find loop invariants for a larger class of programs than the traditional refinement approach and may avoid divergence.

In order to define the class of programs for which our approach is complete, we start with a characterisation of the paths for which our algorithm finds sufficiently strong loop invariants. Given a path π with a loop, where π_0, π_1, and π_2 are loop-free sub-paths, and

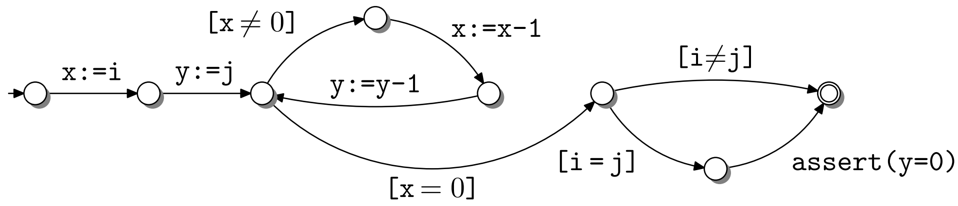
$$\pi \stackrel{\text{def}}{=} \pi_0;\, \|\!:\ [P];\pi_1\ :\!\|\, [\neg P];\pi_2;\mathtt{assert}(Q)\,, \tag{4.6}$$

a predicate R is a sufficiently strong loop invariant if all of the following conditions hold:

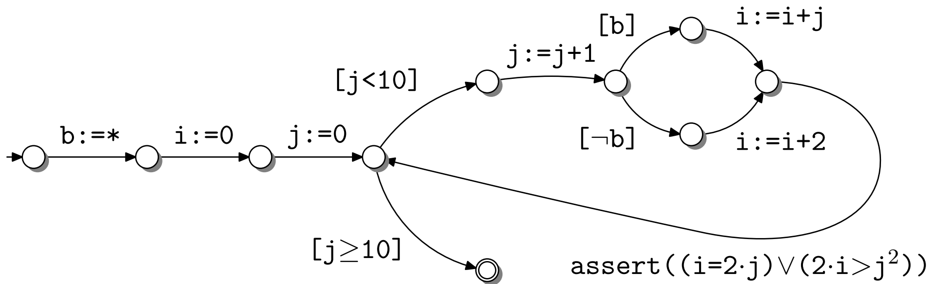
- R is an invariant of the loop, i.e., $\{P \wedge R\}\ \pi_1\ \{R\}$ holds,



(a) Loop body with alternating branches



(b) A program presented by Jhala and McMillan in [JM06]



(c) A loop with a disjunctive invariant

Figure 4.17: A panopticon of programs that can be handled using the approach presented in this chapter

- there is a predicate Q_0 such that $\{\mathsf{true}\}\ \pi_0\ \{Q_0\}$ holds and $Q_0 \Rightarrow R$, i.e., $sp(\pi_o, \mathsf{true})$ implies R and the prefix π_0 establishes the invariant R,
- there is a predicate P_2 such that $\{P_2\}\ \pi_2\ \{Q\}$ holds and $(\neg P \wedge R) \Rightarrow p_2$, i.e., the negated loop condition combined with the invariant R implies the pre-condition of π_2 with respect to the post-condition Q.

These conditions follow immediately from the rule in Figure 4.12. Under the implication order, the loop invariant R is bounded from below by the strongest post-condition of π_0 (denoted by $sp(\pi_0, \mathsf{true})$), and bounded from above by the weakest pre-condition for π_2 terminating with Q true (denoted by $wp(\pi_2, Q)$) (cf. Definition 2.2.1). Note that these bounds are not necessarily loop invariants. Our algorithm is able to prove the path π safe if it manages to compute a loop invariant that lies within these bounds, i.e., the predicates P, which determine the abstract domain, must contain an exact representation of such an invariant.

Our algorithm is succeeds in more cases than the traditional CEGAR approach, since it is able to compute invariants not detected by a refinement approach that does not take the information about loops into account. The class of loop invariants our heuristic is able to infer is restricted, though. Currently, we support only invariants of the form

$$\mathtt{x} = \alpha + \beta \cdot n + \gamma \frac{n \cdot (n+1)}{2} \tag{4.7}$$

(where $\mathtt{x}$ and n are variables and α, β, and γ are expressions constant throughout the loop). Furthermore, an invariant of this kind can only be constructed if the sub-paths π_0 and π_1 of the path π (as defined in (4.6)) match the following pattern:

$$\begin{aligned} \pi_0 &= \ \ldots; \mathtt{x} := \alpha_0;\ \ldots; \mathtt{y} := \alpha_1;\ \ldots \\ \pi_1 &= \ \ldots; \mathtt{x} := \mathtt{x} + \beta_0;\ \ldots; \mathtt{y} := \mathtt{y} + \beta_1 + \mathtt{x};\ \ldots \end{aligned} \tag{4.8}$$

As before, α_0, α_1, β_0, and β_1 are expressions not modified in the loop body, and $\mathtt{x}$ and $\mathtt{y}$ are arbitrary scalar variables. The ellipses indicate arbitrary instructions that do not modify $\mathtt{x}$ and $\mathtt{y}$, and the instructions $\mathtt{y}{:=}\alpha_1$ and $\mathtt{y}{:=}\mathtt{y}+\beta_1+\mathtt{x}$ are optional. Our implementation

simplifies arithmetic expressions in order to increase the number of matches. Furthermore, if the pattern matches more than one set of instructions, the algorithm constructs one invariant for each matching combination of assignments. Since all these invariants hold by construction for the detected loop, this does not lead to a combinatorial explosion.

A syntactic definition more restrictive than the pattern in (4.8) would be too strong: The invariant that is required depends on the assertion that is checked, and a slice of the path may be sufficient to show the safety of the path. Furthermore, we are not restricted to purely arithmetic invariants: If the instructions indicated by the ellipses contain array accesses, pointer arithmetic, or non-linear operations, the resulting loop invariant may use a combination of these theories. If necessary, the set of invariant templates can be increased by using more sophisticated algorithms for solving recurrence equations (see, for instance, [vEBG04]).

We conclude that our algorithm is complete for programs for which all paths with loops are of the form (4.8), and the conditions listed above hold for the resulting invariants. If the sub-paths π_0 and π_2 (see (4.6)) also contain loops, then the conditions have to be strengthened: The upper and lower bounds for the invariant are not determined by the strongest and weakest pre- and postconditions of the paths, respectively, but by the strongest and weakest predicates that the abstraction refinement algorithm can infer at the loop entry and loop exit locations.

The completeness of the traditional abstraction refinement algorithm is analysed and compared to a iterative fixed point algorithm with oracle-guided widening in [BPR02]. The results presented there also apply to our algorithm, since the invariants our algorithm detects are a superset of the invariants detected by the traditional refinement technique.

4.5 Experimental Results

We evaluate our approach using a set of programs that contain known buffer overflows. For this purpose, we implemented the technique described in the previous sections into our predicate abstraction-based verification tool SatAbs [CKSY04, CKSY05][6]. SatAbs

[6] Available at `http://www.cprover.org/satabs/`

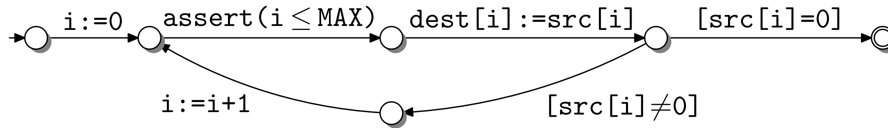


Figure 4.18: A model of the strcpy function

uses a SAT-solver [ES04] as decision procedure. SatAbs translates arithmetic operations into propositional formulas representing the corresponding hardware implementation (cf. Section 3.3), i.e., the bounded size and the bit-vector semantics of integers are modelled accurately. Thus, we rely on efficient satisfiability checking algorithms to solve the constraint-satisfaction problem in step ③ in Figure 4.9. While the SAT-solver may be a potential bottle-neck of the verification process (as indicated at the end of Section 4.3), modern SAT-solvers tend to cope extremely well with problems that do not contain complicated arithmetic expressions.

Table 4.1 shows a comparison of the loop detection algorithm and a version of SatAbs in which the loop detection feature is disabled. We measure the effect of the loop detection algorithm on the number of iterations, the number of predicates generated, and the total runtime of the tool. The experiments discussed in this section were done on an Intel Pentium 4 with 3 GHz and 2 GB of memory. The Aeon benchmark, also presented in [KW06] and listed in the upper section of Table 4.1, demonstrates the potential of our approach: Aeon 0.02a is a mail transfer agent that contains a buffer overflow, which can be triggered by means of an overly long environment variable `HOME`. The content of this variable is obtained using the `getenv` POSIX API function and is copied (by means of `strcpy`) to a buffer of fixed size without checking the length of the string.

We replaced the `getenv` function by a model that returns a string of non-deterministic size and content. Figure 4.18 shows the implementation of `strcpy`, augmented with an assertion[7] that fails if `i` exceeds the upper bound of the string `dest`. In the original program, the size of the `dest` buffer is 512 bytes. Our implementation detects the relevant loop immediately and reports the buffer overflow within 15 seconds, half of which are spent

[7] Our tool SatAbs automatically generates assertions for array bounds, division by zero, and pointer validity. For the Aeon program, which has approximately 800 lines of code, SatAbs generates 576 such assertions.

simulating the unwound counterexample. Without loop detection, SatAbs is unable to provide an answer within a reasonable amount of time. Therefore, we reduced the size of the buffer to 5 and 10 bytes, respectively. As expected, the number of iterations necessary to detect the buffer overflow increases linearly with the size of the buffer. As we report in [KW06], the runtime of traditional predicate abstraction tools grows exponentially with the number of iterations of the loop [KW06], while our loop detection algorithm is not sensitive to the size of the buffer. For a buffer size of 10, our loop detection algorithm is already 400 times faster than the version of SatAbs not supporting the loop detection feature. A similar comparison (also based on Aeon) of our approach to Slam and Blast can be found in [KW06], showing that our heuristic can prevent the exponential increase of the runtime both Slam and Blast exhibit in the presence of loops.

We obtained the remaining entries in Table 4.1 by running our algorithm on problems selected from a buffer overflow benchmark presented by Ku et al. [KHCL07]. These test-cases are simplified versions of a variety of buffer overflow vulnerabilities in open source programs like `OpenSER`, `bind`, and `apache`. We used the unmodified, publicly available benchmark to generate the entries in Table 4.1. The bounds of the loops in these programs are very small, since the benchmark was designed to evaluate software model checking tools that are based on traditional predicate abstraction. If we increase the bounds to a realistic size, the runtime of the traditional algorithm increases exponentially, while the runtime of the loop detection algorithm is mainly determined by the simulation of the unwound counterexample and remains almost unchanged.

The benchmark comprises unsafe programs and their corresponding patched (and therefore safe) counterparts. Our experiments show that our approach works particularly well for unsafe programs, like the `OpenSER` benchmark [KHCL07]. For the corresponding patched version of `OpenSER`, the speedup is less impressive but still measurable: Once the loop is detected, our refinement algorithm adds the relevant predicates in a single iteration. In the case of the `bind` benchmark [KHCL07], our algorithm has no positive impact on the performance. Even though it detects the loop, the number of iterations is too small to result in a significant improvement in runtime. The reason for the performance penalty is that our model checker for abstract programs, in which we implemented the detection

algorithm for abstract loops (see Figure 4.5), is not as fast and optimised as the `SMV` model checker [McM92], which SatAbs uses by default. Our algorithm fails to detect the loop in the patched version of `bind` and in the `apache` benchmark [KHCL07]. In both cases, this has no significant impact on the number of predicates and iterations.

Our experimental results confirm that the technique typically yields a significant performance improvement if our heuristic manages to detect the loop. If, on the other hand, our algorithm fails to detect the crucial loop, the performance impact is negligible.

4.6 Related Work

The approach we present extends the loop detection algorithm in [KW06]. The algorithm presented in [KW06] is covered in Section 4.1. The refinement technique based on adding recurrence predicates (see Section 4.2) is an improvement of the traditional refinement algorithm [BR02a] applied in [KW06].

Beyer et al. [BHMR07b] proposes to combine counterexample-guided abstraction refinement (CEGAR) [CGJ+00] and invariant synthesis to prove the absence of counterexamples. Similar to our approach, the algorithm aims at computing invariants of loops in counterexamples. The resulting *path invariants* contain universal quantifiers. Unlike the recurrence predicates generated by our technique, universally quantified predicates are not readily integrated into existing predicate abstraction-based software verification tools (e.g., Slam [BR02b, BCLR04], Blast [HJMS02], Magic [CCG+04], F-Soft [IYG+05], and SatAbs [CKSY05]) and require special treatment in the abstraction phase. Similar to our recurrence equation-based approach, the algorithm used to generate path invariants is only complete for a certain class of invariant templates (specified in the language of linear arithmetic with uninterpreted function symbols) [BHMR07a]. The class of invariants covered by our approach is discussed in Section 4.4.

Daikon [EPG+07] is a tool that dynamically detects potential ("likely") invariants of a program. It relies on executing the program using a suite of test cases (e.g., regression tests). Daikon traces the values of variables at appropriate points in the program (e.g., procedure entries and exits) by means of instrumenting the code. Using the resulting data,

Application	**Bulletin**	**w/o Loop Detection**		**with Loop Detection**		**Speedup**
		Iterations	**Predicates**	**Iterations**	**Predicates**	
Aeon 0.02a	CVS-2005-1019	>500	>500	2	3	−/15 sec
Aeon 0.02a	CVS-2005-1019 (buffer size 5)	10	54	2	3	6.6
Aeon 0.02a	CVS-2005-1019 (buffer size 10)	23	119	2	3	431.4
OpenSER	CVE-2006-6749	20	99	2	3	171.7
OpenSER	CVE-2006-6749 patched	28	128	20	139	3.9
bind	CVE-2001-0011	7	33	6	28	0.5
bind	CVE-2001-0011 patched	7	45	8	54	0.6
sendmail	CVE-2003-0681	5	33	2	9	3.3
sendmail	CVE-1999-0047	3	8	1	1	1.7
MADWiFi	CVE-2006-6332	5	33	3	27	1.8
apache	CVE-2004-0940	10	55	11	60	1.0

Table 4.1: Experimental results

DAIKON evaluates a large number of potential invariants (applying pre-defined patterns including linear relations over two or three variables, intervals, ordering of sequences, etc.) and reports the invariants that it observes to hold for the recorded test runs. This approach suffers from the same problems as testing, since it depends heavily on the test suite. A single additional test run may invalidate the reported invariants. Thus, the results are not guaranteed to be invariants of the program, though the precision is claimed to be high [EPG+07]. The invariants generated by our approach are guaranteed to be invariants of the analysed paths and not just of a concrete run of that path. Furthermore, DAIKON generates a large number of potential invariants, which makes it unsuitable in our setting, since the scalability of predicate abstraction decreases rapidly with an increasing number of predicates. Our technique generates only invariants that are promising candidates to eliminate a spurious counterexample.

The DAIKON tool is highly extensible and allows the user to add new types of invariants. Our implementation would certainly benefit from such a flexibility. We intend to investigate the feasibility of extending our approach to more complex data-types such as those provided by the C++ template library (see Section 5.2).

Jain et al. [JIG+06] proposes to strengthen the transition relation of the original program using statically computed linear invariants of the form $\pm x \pm y \leq c$. They observe that predicate abstraction generates a more precise abstraction if the original transition relation is strengthened. Since using all generated invariants may not be beneficial, they use a heuristic to filter out invariants not deemed important. In contrast, we compute invariants on demand, and our technique detects a different class of invariants, including nonlinear ones.

Leino and Logozzo suggest to strengthen loop invariants on demand, as the need for stronger invariants arises during the verification process [LL05]. The accuracy of the (numeric) domain used by the abstract interpreter is increased if the theorem prover fails to show the safety of a path. The technique combines invariants generated by means of abstract interpretation with automatic theorem proving. In contrast, our technique and the path invariants approach are based on predicate abstraction and model checking. Furthermore, neither path invariants, nor Leino and Logozzo's or Jain's approach aim at accelerating the

detection of counterexamples.

The incompleteness of traditional predicate-abstraction based CEGAR implementations (e.g., [BR02b, BCLR04]) is a well known problem [Cou00]: If the program is safe, the abstraction refinement algorithm is incomplete unless the refinement step introduces a predicate that represents a (sufficiently strong) invariant of the program. Jhala and McMillan [JM06], to whom we owe the example in Figure 2.10, address this issue by avoiding the generation of a diverging sequence of refinement predicates by restricting the search space of the interpolation-based [HJMM04] refinement algorithm. Unlike our recurrence-based technique and the path invariant approach, their refinement algorithm considers only loop-free counterexamples.

The existence of a counterexample with loops can also be shown by means of induction on the loop bound [WGI07]. This approach has the advantage that it is not necessary to unwind the counterexample. The approach, as presented in [WGI07], is restricted to non-nested loops with a single induction variable and a loop condition that is monotonic with respect to the loop bound. Moreover, it is not suitable for showing the absence of counterexamples.

Path Slicing is an approach that shortens counterexamples by dropping the statements that have no impact on the reachability of the program location in question [JM05]. The statements and branches that can be bypassed are eliminated by backward slicing: For each program location, the set of relevant variables whose valuations at that point determine whether or not the error location is reachable is computed. The feasibility of a path slice implies the feasibility of the original counterexample, but assumes termination of the omitted code sequences. Path slicing eliminates loops during the symbolic simulation if and *only* if they do *not* contribute to the reachability of the error location. Therefore, path slicing is orthogonal to our approach, since it prevents expensive unrolling of loops that are not related to the error.

Ball et al. [BKS07] proposes a technique based on identifying a sequence of must-transitions through loops in an abstract transition system generated by predicate abstraction. In order for this approach to succeed, the concrete transition system must adhere to a set of restrictions, for instance, the abstract state a at the loop entry must represent a

finite set of concrete states, and each concrete state represented by a must not have more than one successor in a. This technique aims at proving the termination of loops in order to leap loops in the abstraction refinement process without the need for further refinement. In contrast, our approach does not impose any restrictions on the concrete transition system. Furthermore, our goal is not proving loop termination, but to find a single counterexample that traverses the loop and violates an assertion.

Linear programs have been proposed by Armando as an alternative, more fine-grained formalism for abstractions of sequential programs [ACM04]. Due to the higher expressiveness of linear programs (in comparison to Boolean programs), this approach yields a smaller number of spurious execution traces. However, the abstraction algorithm is restricted to a pointer-free subset of the C programming language that employs linear arithmetic and arrays [ABM06, ABC$^+$07].

Rybalchenko and Podelski present a complete method for detecting linear ranking functions of non-nested program loops [PR04]. The inferred ranking function poses an upper bound for the iterations of the loop. This bound is not necessarily tight. Combined with abstraction-refinement, this approach enables proofs of program termination [CPR05]. A proof of termination is insufficient to show the feasibility of counterexamples with loops, since the violation of the property usually depends on the number of iterations. Therefore, we utilise a method that provides the exact number of loop iterations necessary to reach the error state.

Acceleration is a technique that aims at computing the repeated iteration of a sequence of transitions of a symbolic transition system in one step [BFLP03, FL02]. It targets finite linear systems and counter automata. The technique accelerates the computation of the reachable states of the system, but does not specifically target the detection of counterexamples. Acceleration is also called exact widening [CC77]. Our heuristic GUESSITERATIONS(π) (see Figure 4.9) may also be interpreted as a widening and acceleration step on the transition function defined by the body of the detected loop.

Chapter 5

Future Directions and Conclusion

Our dissertation presents novel techniques for interpolation-based software model checking, an approximate method which uses Craig interpolation [Cra57a, Cra57b] to construct abstractions. We consider two aspects of program analyses based on model checking: verification (the construction of Hoare proofs [Hoa69] for programs) and falsification (the detection of counterexamples that violate the specification).

A Hoare proof comprises assertions and loop invariants which serve as a certificate of the correctness of a program with respect to a given specification. Chapter 2 provides an overview of state-of-the-art techniques aimed at computing such proofs by means of symbolic simulation and automated abstraction. The principal challenge is to select appropriate assertions and loop invariants from a range of potential candidates.

Craig interpolation is a promising technique to aid this selection. A Craig interpolant can be derived from a symbolic representation of an execution trace that does not violate the specification. The resulting interpolant represents a set of states which is (a) an upper estimation of the states reachable along the trace and (b) a lower estimation of the *safe* states from which no "bad" states can be reached along the trace. The construction of a Hoare proof is based on iteratively considering execution traces and their corresponding interpolants until an invariant establishing the safety of the program is found. The success of this approach depends on whether the resulting sequence of interpolants converges.

Convergence, in turn, inherently depends on the choice of interpolants. The work of Ranjit Jhala [Jha04] demonstrates that the use of Craig interpolants leads to a better

performance than traditional techniques such as Dijkstra's predicate transformers [Dij75] (within the same verification framework). McMillan's results in [McM06] corroborate this observation. The improved performance is attributed to the fact that the abstraction obtained using Craig interpolation is more "parsimonious" [HJMM04].

Both implementations, however, are based on an interpolation technique that is entirely *ad-hoc* and yields only a single interpolant [McM05], affording no other choice of assertions for the respective execution trace. The same restriction applies to other recent interpolation procedures (for instance [KMZ06, KW07, BZM08, FGG$^+$09, GKT09, KV09b, KW09a, BKRW10]).[1] The interpolation techniques presented in Chapter 3 lift this restriction: we show that the logical strength and the symbols occurring in the interpolants can be systematically tuned. The preliminary experiments in Section 3.6 show that the impact of interpolant strength on the performance of the verification tool cannot be neglected.

Moreover, our interpolation techniques target three different levels of abstraction. The interpolation system discussed in Section 3.3 extracts propositional interpolants from bit-flattened representations of symbolic execution traces, targeting the lowest level of abstraction. Section 3.5 covers interpolation systems for word-level formulae. These systems, while sound with respect to the bit-level semantics of software programs, yield interpolants which preserve the integrity of variables and word-level operations encountered in the program. This approach maintains a higher level of abstraction than propositional interpolation systems. Section 3.4 presents a technique which enables us to combine bit-level and word-level interpolation systems. Finally, the approach discussed in Section 4.2 takes the control-flow structure of programs into account. This approach is orthogonal to the interpolation techniques presented in Chapter 3. We exploit the information about repetitive programming constructs in order to accelerate the convergence of the sequence of the interpolants obtained by exploring execution traces of the program.

The information about loops in execution traces proves particularly advantageous in the context of program falsification. We present a technique to obtain and exploit this information in Chapter 4. Equipped with the information about repetitive constructs in

[1] The interpolation procedures presented in [FGG$^+$09, GKT09, KW09a] leave some room for variation in the colour of a limited number of partial interpolants, c.f. Section 3.5.

an execution trace, we construct a parametric representation of the trace which encodes arbitrarily many repetitions. We then use a decision procedure to examine the parametrised trace for counterexamples. Our technique helps to avoid the repeated and computationally expensive construction of interpolants for structurally similar execution traces if one of these traces violates the specification. It enables the detection of counterexamples that contain deep loops. A typical example of program defects demonstrated by such counterexamples are buffer overflows. The experimental evaluation in Section 4.5 shows that our approach enables the detection of bugs of this nature: our implementation outperforms the traditional abstraction-refinement approach based on predicate abstraction on many typical buffer overflow examples.

5.1 Open Issues

While Section 3 presents a range of different interpolation systems which allow for variation of the strength and structure of the interpolants, it leaves the question of how to choose a promising candidate for the invariant open. The experiments in Section 3.6 show that the impact of interpolant strength on the performance of the verification tool cannot be neglected. At this point, however, we fail to provide a method to determine whether it is beneficial to strengthen or weaken interpolants: our experiments show that neither choice is consistently superior. We believe that this problem can only be solved empirically. A wider evaluation based on a more diverse set of programs is required to provide a more sophisticated analysis of the impact of interpolant strength. We defer this detailed analysis to future work.

5.2 Future Directions

The experimental results in Chapter 4 indicate that an *informed* refinement approach which exploits information about the program structure can enable a significant improvement of the performance of a verification tool. The interpolation-based refinement technique presented in Section 2.5.3, however, considers only single execution traces. Moreover, while the interpolation systems presented in Chapter 3 enable the construction of a wider range

of Craig interpolants, we currently lack the means to guide the selection of interpolants. We conjecture that factors other than interpolant strength may have a significant impact on the performance of the tool. For instance, Theorem 3.3.1 enables a systematic inclusion or exclusion of word-level atoms in interpolants. It is possible to guide the construction of interpolants using external information about the program provided by the model checking algorithm. Information about the program structure or a computationally inexpensive static analysis can aid the choice of potential invariants (see, for instance, [KV09a] and [JIG+06], respectively).

In many software verification tools (e.g., Slam [BCLR04], Blast [HJMS02], SatAbs [CKSY05], and Wolverine) the underlying interpolating decision procedure is only loosely coupled with the component performing the static analysis. The recent advances of SMT solvers led to verification tools (such as CPAchecker [BCG+09]) which delegate more work to the decision procedure by encoding control flow constructs (such as conditional statements) in the formula presented to the solver. While this approach enables us to encode a large number of execution traces in a single formula, the decision procedure is not *aware* of the semantics of control flow constructs. Therefore, it cannot take advantage of potentially available information about the control flow structure of the program. In the future, we expect to see interpolating decision procedures which are more tightly integrated into the static analysis.

Furthermore, we intend to investigate the impact of interpolant properties other than logical strength and structure. For instance, we conjecture that proof restructuring techniques (as presented in [BIFH+09]) and an opportunistic choice of the colour of conclusions in refutation proofs (c.f. Section 3.5) enable a reduction of the size of interpolants. A first step in this direction is reported in [FGG+09]. While the size of interpolants is not an immediate problem in our setting, it may become a paramount factor in the presence of more complex transition functions (which typically derive from hardware models or may arise from the encoding of control flow constructs presented in [BCG+09]).

Finally, our idea to exploit the information about the control flow structure of a program in order to obtain "better" interpolants is not restricted to the simple invariants discussed in Section 4.4. We plan to extend our approach to invariants over more complicated data-

types such as those provided by the C++ template library. We intend to base this work on the verification technique presented in [BGK07], which is based on an abstract model of the template library that preserves relevant facts such as the size of C++ containers (e.g., lists). Furthermore, we have plans to integrate our approach into our interpolation-based model checking tool WOLVERINE.

The idea of extracting invariants from paths is very promising and has recently been successfully applied in a number of different ways (e.g., see [KW06, BHMR07b, EPG+07], discussed in Section 4.6). It is particularly powerful in combination with a refinement-based static analysis technique, allowing it to derive non-trivial disjunctive invariants. We expect to see extensions of our idea that enable the verification of a larger, more general class of programs.

5.3 Conclusion

Our dissertation discusses interpolation algorithms which afford us a choice of different interpolants. Furthermore, our interpolation systems are sound with respect to the bit-vector semantics of software programs. Finally, we present a technique which exploits information about the presence of loops in order to accelerate the detection of counterexamples or to derive interpolants for a family of safe execution traces in a single step. Each single contribution is, to the best of our knowledge, an improvement over the state-of-the-art. The experimental evaluation of the implementation of the techniques presented in this dissertation demonstrates their potential to improve the performance of verification tools. While a further investigation of the impact of a different choice of interpolants is required, the promising experimental results suggest that a further pursuit of this line of research is worthwhile.

Appendix A

Implementation

A.1 Implementation Details of Wolverine

Our interpolation-based software verification tool WOLVERINE is an implementation of the algorithm presented in [McM06]. It is based on Daniel Kroening's CPROVER framework,[1] which provides a mature and robust front-end for ANSI-C and C++ programs and a bit-level accurate symbolic simulator for these programming languages.

WOLVERINE aims at constructing the complete complete reachability tree (see Definition 2.5.1 in Section 2.5.1) of a program. To this end, it unwinds the control-flow graph (see Section 2.3) in a depth-first search manner. Each path in this graph which reaches an assertion is a potential counterexample. The feasibility of the corresponding program paths is checked using the symbolic simulator of the CPROVER framework, which generates an SSA representation of the respective strongest postcondition (see page 29 in Section 2.1). The simulator uses the SAT solver MINISAT [ES04] to decide the satisfiablity of the formulae (generated by the bit-flattening approach discussed in Section 3.3.1) representing program paths.

Spurious counterexamples are eliminated by refining the parametrised predicate transformer (defined at the end of Section 2.2). We rely on the interpolation-based refinement technique discussed in Section 2.5.3. The underlying interpolating decision procedure is

[1]The CPROVER framework comprises the components the verification tools CBMC [CKL04] and SATABS [CKSY05] are based on.

described in Section A.2. An algorithmic description of our interpolation technique is presented in [KW09a]. The algorithm there is essentially an implementation of the interpolation system described in Section 3.5. We deal with the propositional structure of formulae using the proof lifting technique introduced in Section 3.4 (which is based on the idea presented in [KW07]): We use MiniSAT to find an unsatisfiable core of an inconsistent formula and use the simple SMT-algorithm presented in Section 3.4.1 to construct a refutation of the conjunction of the propositional skeleton and the blocking clauses.

Wolverine is available from the website `http://www.cprover.org/wolverine`.

A.2 A Graph-Based Decision Procedure

This section describes a graph-based decision procedure for the conjunctive fragment of bit-vector arithmetic formulae. The section contains excerpts of our publication [KW09a].[2]

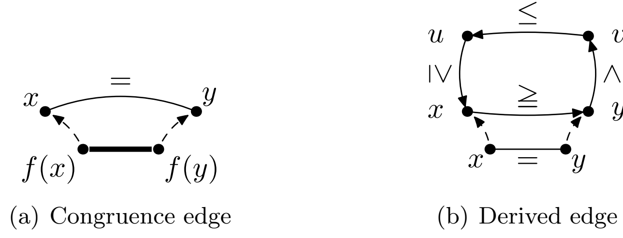
The grammar for bit-vector formulae is specified in Table 3.1 in Section 3.1, and the restricted conjunctive fragment is defined in Table 3.5 in Section 3.5.1. In the following, we explain how formulae in this logic (to which we refer to as $\mathcal{L}$ in this section) can be represented as graphs.

Graph representation of $\mathcal{L}$ formulae. The fact that an $\mathcal{L}$-formula F is a conjunction of atoms of the form $t_i \rhd t_j$ enables us to represent F using a graph [MS05].

Definition A.2.1 ($\mathcal{L}$-graph)**.** *Given a formula F, let $\mathcal{G}_F(V, E)$ be a directed graph, where each term t_i in F corresponds to a node v_i in V, and each atom $t_i \rhd t_j$ corresponds to a $\rhd$-labelled edge $(v_i \stackrel{\rhd}{\rightarrow} v_j) \in E$, $\rhd \in \{=, \geq, >, \neq\}$. Atoms $t_i \rhd t_j$ with a* symmetric *relation $\rhd \in \{=, \neq\}$ additionally contribute an edge $(v_j \stackrel{\rhd}{\rightarrow} v_i)$. For convenience, we use undirected edges to depict equalities and disequalities. In accordance to [McM05], we write $v_i \simeq v_j$ if and only if $i = j$.*

Due to the presence of functions in $\mathcal{L}$-terms, the congruence rule in Table 3.7 may give rise to additional equality edges in the graph: The congruence relation satisfies, in

[2]With kind permission from Springer Science+Business Media: Haifa Verification Conference, An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions, October 2009, Daniel Kroening and Georg Weissenbacher, © 2009-2010



(a) Congruence edge (b) Derived edge

Figure A.1: Congruence edges and derived edges

addition to the properties of the equality relation, the monotonicity axioms, i.e., for all n-ary functions f, it holds that $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ whenever $s_i = t_i$ holds for all i in $\{1, .., n\}$. We use *congruence edges* to depict such equalities (see Fig. A.1(a)). The dashed arrows indicate that $f(x) = f(y)$ is derived from the equality of the sub-terms $x = y$.

Definition A.2.2 (Contradictory and equality-entailing cycles)**.** *A* contradictory cycle *in an $\mathcal{L}$-graph is a cyclic path consisting of either*

a) edges labelled with $=$ and a single edge labelled with $\neq$, or

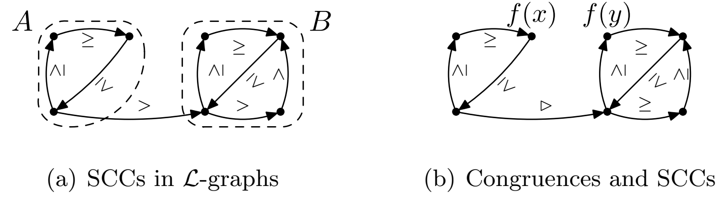
b) edges labelled with either $=$ or $\geq$ and at least *one edge labelled with $>$.*

An equality-entailing cycle *in an $\mathcal{L}$-graph is a cyclic path consisting of edges labelled with either $=$ or $\geq$. For any two terms t_i and t_j corresponding to nodes in an equality-entailing cycle, it holds that $t_i \geq t_j$ and $t_j \geq t_i$, and thus $t_i = t_j$.*

We depict derived edges using a graphical representation similar to congruence edges (see Fig. A.1(b)). In this example, the equality $x = y$ is derived from the equality-entailing cycle $x \stackrel{\geq}{\rightarrow} y \stackrel{\geq}{\rightarrow} v \stackrel{\geq}{\rightarrow} u \stackrel{\geq}{\rightarrow} x$.

We provide a brief outline of our decision procedure for $\mathcal{L}$-formulæ followed by a detailed description of the proof-generating algorithm. Let $\mathcal{G}(V, E)$ be the $\mathcal{L}$-graph for a given formula F. The decision procedure is subdivided into two phases, which are repeatedly iterated until the formula can be refuted or until no new facts can be derived:

1. In the first phase, the algorithm searches for contradictory or equality-entailing cycles with edges labelled $=$, $\geq$, and $>$ (Def. A.2.2a) in the graph $\mathcal{G}(V, E^{\stackrel{\geq}{=}})$, where $E^{\stackrel{\geq}{=}}$ denotes $E \setminus \{(v_i \stackrel{\neq}{\rightarrow} v_j) \in E\}$. If a contradictory cycle exists, the algorithm terminates.



(a) SCCs in $\mathcal{L}$-graphs (b) Congruences and SCCs

Figure A.2: Strongly connected components (SCCs) in $\mathcal{L}$-graphs

Otherwise, the procedure adds to E the edges $v_i \xrightarrow{=} v_j$ and $v_j \xrightarrow{=} v_i$ for all nodes v_i, v_j adjacent in an equality-entailing cycle.

2. In the second phase, additional equalities are inferred by means of constant propagation and congruence closure and searches for contradictory cycles with edges labelled $=$ or $>$ (Def. A.2.2*b*) in the graph $\mathcal{G}(V, E^{\neq})$, where $E^{\neq} = \{(v_i \rhd v_j) \in E \mid \rhd \in \{=, \neq\}\}$.

The phases are iterated until no new equalities can be inferred. Both phases use well-known and efficient graph algorithms such as Tarjan's algorithm for the computation of *strongly connected components* (SCCs) and a graph-based *union-find* data structure. In a pre-processing step, we form two (possibly non-disjoint) sets of the atoms in F, one of which contains the inequalities and equalities, and one which contains equalities and disequalities.

Phase I: Inequalities. Let $\mathcal{G}(V, E^{\geqq})$ be the $\mathcal{L}$-graph corresponding to the equality and inequality atoms of F. Using Tarjan's algorithm, we compute all strongly connected components in $\mathcal{G}(V, E^{\geqq})$ and classify them as contradictory or equality-entailing cycles, respectively:

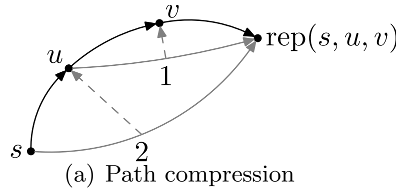
1. A SCC is contradictory if it contains at least one edge $v_i \xrightarrow{>} v_j$ (see component B in Fig. A.2(a)). Then, any path from v_j to v_i forms a contradictory cycle with $v_i \xrightarrow{>} v_j$. If our algorithm finds a contradictory SCC, we compute the shortest such path and report it as a proof of inconsistency.

2. A SCC is equality-entailing if it contains no edge labelled with $>$ (see component A in Fig. A.2(a) or the SCCs in Fig. A.2(b)). In this case, we conclude that for any edge $v_i \xrightarrow{\rhd} v_j$ in the SCC $t_i = t_j$ holds for the corresponding terms. The derived equalities are passed on to the second phase.

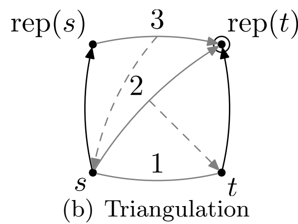
Phase II: Equalities and Disequalities. The second phase starts with computing the equivalence closure of the equality atoms (and the equalities derived in the first phase). For this purpose, we use a *proof-generating* union-find data structure that incrementally constructs an $\mathcal{L}$-graph $\mathcal{G}(V, E^{=})$, where $E^{=}$ denotes a set of edges labelled with =. In the following, we present the modifications necessary to generate a proof of inconsistency. In a union-find data structure, each equivalence class corresponds to a sub-graph of $\mathcal{G}(V, E^{=})$ identified by its *representative*, and each node which is not a representative holds a reference to its parent node (indicated by an directed edge in our illustrations). The data structure supports two operations:

1. $\mathit{Find}(v_i)$ returns the representative of the node v_i.

2. $\mathit{Union}(v_i, v_j)$ adds an (undirected) equality edge to the graph $\mathcal{G}(V, E^{=})$ and merges the two equivalence classes containing v_i and v_j, respectively.



(a) Path compression



(b) Triangulation

assert $v_1 \not\simeq v_2 \wedge r_1 \not\simeq r_2$
if $v_1 \simeq r_1$ **then**
  if $v_2 \not\simeq r_2$ **then**
    $E^{=} := E^{=} \cup \{r_2 \overset{\langle v_2 \rangle}{\rightarrow} r_1\}$
  end if
else
  if $v_2 \not\simeq r_1$ **then**
    $E^{=} := E^{=} \cup \{v_2 \overset{\langle v_1 \rangle}{\rightarrow} r_1\}$
  end if
  if $v_2 \not\simeq r_2$ **then**
    $E^{=} := E^{=} \cup \{r_2 \overset{\langle v_2 \rangle}{\rightarrow} r_1\}$
  end if
end if

(c) Implementation of A.3(b)

Figure A.3: An illustration of union-find operations

The $\mathit{Find}(v_i)$ operation performs *path compression* in order to reduce the computational effort in case of repeated queries for v_i. During this process, it adds new derived edges to $E^{=}$, which connect v_i directly with its representative. This is illustrated by the example in Fig. A.3(a). *Find* follows the parent nodes until it reaches the representative. In Fig. A.3(a),

the call to $\mathit{Find}(v_1)$ results in two recursive calls $\mathit{Find}(v_2)$ and $\mathit{Find}(v_3)$. The latter call returns v_4 as the representative for v_3. We add $v_2 \overset{\langle v_3 \rangle}{\rightarrow} v_4$ to $E^=$ (step 1 in Fig. A.3(a)) and replace the parent v_3 with v_4. Here, the label $\langle v_3 \rangle$ is used to memorise the fact that $v_2 \overset{=}{\rightarrow} v_3$ derives from $v_2 \overset{=}{\rightarrow} v_3 \overset{=}{\rightarrow} v_4$ (visualised by the dashed arrow). Finally, $\mathit{Find}(v_2)$ yields v_4 and we add $v_1 \overset{\langle v_3 \rangle}{\rightarrow} v_4$ to $E^=$ and replace the parent v_2 with v_4. Thus, $\mathit{Find}(v_1)$ returns v_4.

The $\mathit{Union}(v_1, v_2)$ operation merges two equivalence classes with the representatives r_1, r_2 (obtained using *Find*). We assume that redundant unions are ignored, i.e., $v_1 \not\simeq v_2$ and $r_1 \not\simeq r_2$. Consider the example in Fig. A.3(b). We add the edge $v_1 \overset{=}{\rightarrow} v_2$ (step 1) and conclude that the terms corresponding to r_1 are r_2 equivalent. The algorithm chooses a new representative (r_1 in our example), favouring nodes with a higher in-degree. The resulting edge $v_2 \overset{\langle v_1 \rangle}{\rightarrow} r_1$ is labelled accordingly in step 2, in order to memorise its derivation. Finally, we connect r_2 and r_1; the corresponding edge derives from $r_2 \overset{=}{\rightarrow} v_2 \overset{=}{\rightarrow} r_1$.

Observe that *Union* triangulates the sub-graph spanning $V = \{v_1, v_2, r_1, r_2\}$. Fig. A.3(c) shows the general algorithm for this triangulation (where r_1 is the representative node with the higher in-degree), which is a constant time operation.

Using *Union*, we compute the equivalence closure for F by adding all equivalence atoms and derived equalities to $\mathcal{G}(V, E^=)$. We can now efficiently query whether a disequality $t_i \neq t_j$ contradicts the equality relations stored in $\mathcal{G}(V, E^=)$ by checking whether $\mathit{Find}(v_i) \simeq \mathit{Find}(v_j)$. If this is the case, we obtain a contradictory cycle $v_i \overset{\neq}{\rightarrow} v_j \overset{=}{\rightarrow} r \overset{=}{\rightarrow} v_i$. From this cycle, we obtain a proof for the inconsistency by repeatedly expanding derived edges $v_i \overset{\langle v_j \rangle}{\rightarrow} v_k$ to $v_i \overset{=}{\rightarrow} v_j \overset{=}{\rightarrow} v_k$. Edges derived in Phase I are justified by their respective equality-entailing cycles.

Congruence closure. The decision procedure described above lacks a provision for deriving congruence edges (Fig. A.1(a)) and is therefore not sufficient to support uninterpreted functions. An equality relation $t_i = t_j$ in the congruence graph $\mathcal{G}(V, E^=)$ gives rise to a congruence edge representing $f(t_i) = f(t_j)$, which, in return, may entail additional equality relations in $\mathcal{G}(V, E^=)$. Therefore, we use an incremental *congruence closure* algorithm (following the ideas presented in [NO05]) that is closely intertwined with the construction of the $\mathcal{L}$-graph for equalities.

The algorithm uses the union-find data-structure representing $\mathcal{G}(V, E^{=})$. It *indexes* each representative in $\mathcal{G}(V, E^{=})$ with a term t_c. Thus, all terms in an equivalence class of $\mathcal{G}(V, E^{=})$ are associated with the same term t_c. If the equivalence class contains an interpreted constant (e.g., a numeral), we choose it as the index term,[3] otherwise, we use the term corresponding to the representative of the equivalence class as index. In this setting, an equivalence class containing terms with function symbols represents a set of congruence relations. Consider two terms $f(t_i)$ and $f(t_j)$, where t_i and t_j have the same representative indexed with t_c. Then $f(t_i)$ and $f(t_j)$ belong to the same equivalence class.

In addition, we maintain a function $\mathit{Lookup}(f(t_c))$ which maps $f(t_c)$ to a term $f(t_i)$ such that t_i belongs to an equivalence class indexed with t_c, or $\perp$ if there is no such term in $\mathcal{G}(V, E^{=})$.

The *Union* operation potentially changes the representatives of the equivalence classes in $\mathcal{G}(V, E^{=})$. Therefore, the algorithm maintains for each index term t_c a list *UseList*(t_c) of terms that contain a sub-term indexed with t_c. This list is updated whenever *Union* merges two equivalence classes. W.l.o.g., assume that *Union* merges an equivalence class indexed with t_c with an equivalence class indexed with t'_c, choosing the latter term as the new index. Then, for each $f(t_i) \in \mathit{UseList}(t_c)$, where t_c is the index term associated with t_i, the algorithm proceeds as follows:

- If $\mathit{Lookup}(f(t'_c))$ returns $f(t_j)$, it uses *Union* to add a the congruence edge for $f(t_i) = f(t_j)$ to $\mathcal{G}(V, E^{=})$ and memorises that the edge is derived from $t_i = t_j$. Furthermore, $f(t_i)$ is moved from *UseList*(t_c) to *UseList*(t'_c).
- If $\mathit{Lookup}(f(t'_c))$ returns $\perp$, it sets $\mathit{Lookup}(f(t'_c))$ to $f(t_i)$ and moves $f(t_i)$ to *UseList*(t'_c).

Example A.2.1. *Consider a union-find data structure with four equivalence classes* $\{f(z)\}$, $\{f(x), f(y)\}$, $\{x\}$, *and* $\{z\}$ *(see Fig. A.4, on the left).* UseList$[z]$ *contains* $f(z)$, *since* z *is a sub-term of* $f(z)$. *Adding* $x = z$ *yields a new equivalence class* $\{x\} \cup \{z\}$. *Assume that the representative of the resulting equivalence class* $\{x, z\}$ *is* x *and that* $\mathrm{Lookup}(f(x)) = f(y)$. *Then the algorithm infers* $f(z) = f(y)$.

[3] Note this constant is unique, since an equivalence class that contains two constants with a different interpretation contains a contradictory cycle.

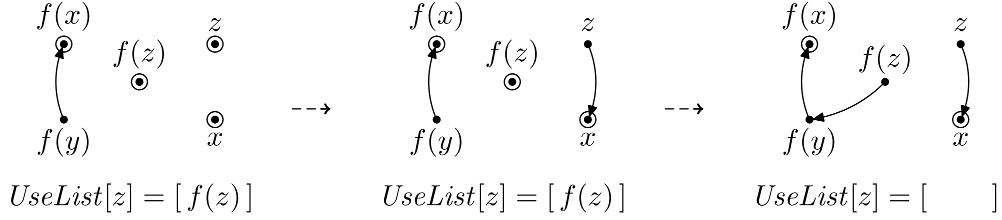


Figure A.4: A 3-step example illustrating the congruence closure algorithm

The extension to n-ary functions is straight-forward. An efficient implementation based on *currifying* is presented in [NO05].

Bit-vector theory axioms, constant propagation, and interpreted functions. Our decision procedure provides limited support for the theory of bit-vectors by integrating a small set of bit-vector inference and axiom instantiation rules. Furthermore, whenever possible, it uses interpreted functions and constants in order to simplify terms. This is achieved by the following mechanisms:

1. We order all interpreted constants $c_1, \ldots, c_n$ processed in Phase I and add $n-1$ inequality relations of the form $c_i < c_{i+1}$, $1 \leq i < n$ to $\mathcal{G}(V, E^{\gtreqqless})$ *before* computing the SCCs.

2. In Phase II, if *Union* is applied to two terms indexed with different interpreted constants c_1 and c_2, we introduce the disequality $c_1 \neq c_2$.

3. Let T be the set of terms corresponding to the nodes in $\mathcal{G}(V, E^{=})$. For each $f(t_i) \in T$ such that f is an interpreted function symbol in a given theory $\mathcal{T}$ and t_i is a term indexed with an interpreted constant c, we check whether $f(c)$ can be simplified to a term t_j not containing any variables or function symbols that do not occur in $f(c)$. If this is the case, and $t_j \in T$ or t_j is an interpreted constant, we add the equivalence relations $f(t_i) = f(c)$ (derived from $t_i = c$) and $f(c) = t_j$ (a tautology in $\mathcal{T}$) to $\mathcal{G}(V, E^{=})$. This technique allows us to perform *bit-level-accurate* simplifications of terms.

4. We instantiate a fixed set of axioms of the form $t \rhd t'$ if we encounter the term t, where t' is the term obtained by instantiating the axiom for t. All rules have the property

$$
\begin{array}{llll}
(t_2 + c) \neq t_2 & \text{if } c \neq 0 \bmod 2^m & (t \ll c) = (t + t) & \text{if } c = 1 \\
(t_2 + c) = t_2 & \text{if } c = 0 \bmod 2^m & (t \ll c) = (2^c \cdot t) & \text{if } 1 < c < m
\end{array}
$$

Figure A.5: Two examples for axioms for m-bit variables

$$
\frac{t_1 = t_2 \,\&\, t_3}{t_1 \leq t_2 \quad t_1 \leq t_3} \qquad \frac{t_1 = t_2 \,|\, t_3}{t_1 \geq t_2 \quad t_1 \geq t_3} \qquad \frac{t_1 + t_2 = t_1}{t_2 = 0}
$$

Figure A.6: Examples of rules for bit-vector operations for unsigned integers

that they do not introduce variables. Examples of such axioms are listed in Fig. A.5. If t and t' correspond to nodes in $\mathcal{G}(V, E^=)$, we add the relation $t \rhd t'$.

5. Inference rules of the form $(t_1 \rhd_1 t_2) \vdash (t_3 \rhd_2 t_4)$ may be applied if t_3 and t_4 refer to a subset of the non-logical symbols in t_1 and t_2. Examples of such inference rules are provided in Fig. A.6 and Table 3.7.[4]

Array accesses. We provide limited support for arrays based on the extensionality rule, which defines the equality of two arrays as element-wise equality by means of universal quantification. In accordance with [KS08], we use $\mathtt{a}\{\mathtt{x} \leftarrow \mathtt{y}\}$ to denote the array $\mathtt{a}$ where the element with index $\mathtt{x}$ has been replaced with $\mathtt{y}$. Array updates $\mathtt{a}[i]{:=}t$ are then handled by introducing an assignment $\mathtt{a}_{m+1}{:=}\mathtt{a}_m\{i \leftarrow t\}$ in SSA form (cf. page 29), which in turn is encoded as

$$
\mathtt{a}_{m+1}[i] = t \;\wedge\; \forall j \neq i \,.\, \mathtt{a}_{m+1}[j] = \mathtt{a}_m[j]\,. \tag{A.1}
$$

Since our decision procedure does not support universal quantification, we instantiate unquantified equalities on demand. If the terms $\mathtt{a}_{m+1}[n]$ and $\mathtt{a}_m[n]$ occur in the set of terms T which corresponds to the nodes in $\mathcal{G}(V, E^=)$ when we encounter the fact (A.1), we determine the representatives of the terms n and i. If these representatives are indexed with two different constants, we conclude that $n \neq i$ and instantiate $\mathtt{a}_{m+1}[n] = \mathtt{a}_m[n]$ according to the rule in Figure A.7. Furthermore, we treat each array access $\mathtt{a}[t]$ as an uninterpreted function a with the parameter t (i.e., as term $a(t)$).

[4]The naïve application of such axioms increases the complexity of the algorithm significantly. Therefore, we apply each axiom only once in an initial rewriting phase.

$$\frac{\mathsf{a}_{m+1}[i] = t \wedge \forall j \neq i \,.\, \mathsf{a}_{m+1}[j] = \mathsf{a}_m[j] \qquad n \neq i}{\mathsf{a}_{m+1}[i] = t \qquad \mathsf{a}_{m+1}[n] = \mathsf{a}_m[n]} \; (\mathsf{a}_{m+1}[n], \mathsf{a}_m[n] \in T)$$

Figure A.7: Quantifier instantiation for arrays

Combining both phases. As explained above, equality relations derived from equality-entailing cycles in Phase I are passed on to Phase II. Now consider the $\mathcal{L}$-graph in Fig. A.2(b). Adding the congruence edge corresponding to $f(x) = f(y)$ results in a new SCC, which, depending on the label $\triangleright$ in Fig. A.2(b), is either contradictory or equality-entailing. Therefore, the congruence edges generated in Phase II must be added to $\mathcal{G}(V, E^{\geqq})$, necessitating an additional iteration of Phase I. The two phases need to be iterated until no more new congruence edges are generated. Since both phases are exchanging equalities exclusively, our implementation is essentially a Nelson-Oppen-style decision procedure.

Complexity. Tarjan's algorithm applied in Phase I has a run-time linear in the number n of edges of the graph. The computation of the equivalence closure in the second phase takes $O(n \cdot \alpha(n))$ time, where α is the inverse of the Ackermann function $A(n, n)$. The congruence closure is of complexity $O(n \cdot \log n)$ [NO05]. Thus, a single iteration of Phase I and Phase II takes $O(n \cdot \log n)$ time.

It remains to determine how often the phases need to be iterated. Since the algorithm never adds redundant congruence edges, the congruence closure adds at most $O(n)$ equalities (see [NO05]). Due to the restrictions on the application of rules and axioms, rewriting interpreted functions increases the number of sub-terms by at most a constant factor. Altogether, we face a run-time complexity of $O(n^2 \cdot \log n)$ for our decision procedure.

Finally, the extraction of an explanation from a contradictory cycle can be performed in $O(n \cdot \log n)$ time, since the derived edges form a tree.

Proofs of inconsistency. We review the artefacts generated by our decision procedure. A proof of inconsistency of an $\mathcal{L}$-formula F is a *contradictory cycle* comprising

- edges directly corresponding to relations in F,
- edges derived from equality-entailing cycles, and

- congruence edges, derived from a number of equality relations.

Analogously to the colouring of formulae (Definition 3.5.2), we introduce the concept of coloured $\mathcal{L}$-graphs.

Colouring $\mathcal{L}$-graphs Given an $\mathcal{L}$-formula $F \wedge G$, we say that a node v_i of the corresponding graph $\mathcal{G}(V, E)$ is F-colourable if the corresponding term t_i refers only to uninterpreted symbols in F; similarly for G. We use V_F and V_G to refer to the set of F-colourable and G-colourable nodes, respectively. This definition splits $V = V_F \cup V_G$ into two non-disjoint sets of vertices. It leaves us a choice for a subset $V_S \stackrel{def}{=} (V_F \cap V_G)$ of V. We refer to V_S as *shared vertices.*

An edge $v_i \stackrel{\rhd}{\rightarrow} v_j$ is F-colourable if and only if $\{v_i, v_j\} \subseteq V_F$; analogously for G. We use E_F (E_G) to refer to the F-colourable (G-colourable, respectively) edges in E. An edge is colourable if it is either F-colourable or G-colourable. The edges of the initial $\mathcal{L}$-graph $\mathcal{G}(V, E)$, in which each edge corresponds to an atom in $F \wedge G$, are always colourable. This is not necessarily the case for the graph that we obtain by computing the congruence closure (in Phase II). Consider the nodes labelled $f(x)$ and $f(y)$ in the $\mathcal{L}$-graph in Fig. A.2(b). Assume that the variable x occurs only in F and y occurs only in G. If we deduce $f(x) = f(y)$ from $x = y$, then the corresponding edge is not colourable.

It is, however, possible to transform a congruence-closed $\mathcal{L}$-graph into a colourable graph [FGG+09, YM05]. We provide a constructive proof based on structural induction over an $\mathcal{L}$-graph with congruence edges:

1. *Base case.* Colour the equality edges of the $\mathcal{L}$-graph according to their respective atoms in the formula $F \wedge G$.

2. *Induction step.* The argument is split into two cases:

   (a) *Derived edges.* For each edge $v_i \stackrel{=}{\rightarrow} v_j$ derived from an equality-entailing cycle, there exists an edge $v_i \stackrel{\rhd}{\rightarrow} v_j$ ($\rhd \in \{\geq, =\}$) in that cycle, which is, by the induction hypothesis, colourable. Let $v_i \stackrel{=}{\rightarrow} v_j$ take the colour of that edge.

   (b) *Congruence edges.* Pick any non-colourable congruence edge with nodes $v_{f(x)}$ and $v_{f(y)}$ labelled $f(x)$ and $f(y)$, respectively. By the induction hypothesis, all

edges in the path $v_x \to \ldots \to v_y$ entailing $x = y$ can be coloured. Since v_x and v_y are of different colour, there is a path prefix $v_x \to \ldots \to v_z$ such that all nodes in the prefix are of the same colour and $v_z \in V_S$. Let z be the term that corresponds to v_z. Then, the term $f(z)$ refers only to non-logical symbols common to F and G. Introduce a new node $v_{f(z)}$ representing $f(z)$ and add an equality edge $v_{f(x)} \to v_{f(z)}$ justified by $v_x \to \ldots \to v_z$, and a new congruence edge $v_{f(z)} \to v_{f(y)}$ justified by $v_z \to \ldots \to v_y$. All these new elements are colourable.

This proof translates into an algorithm of complexity $O(n \cdot \log n)$. The transformation yields a graph representing a formula *equisatisfiable* with $F \wedge G$, i.e., the modified graph contains a contradictory cycle if and only if the original congruence-closed graph $\mathcal{G}(V, E)$ contains one.

It is straight-forward to extend this argument to the edges introduced by the term rewriting rules and axioms in Section A.2. Consider, w.l.o.g., an F-coloured node v_i corresponding to a term t, and a node v_j corresponding to the rewritten term t'. Due to the restriction that the rewriting rule $t \rightsquigarrow t'$ must not introduce new non-logical symbols,[5] the edge $v_i \to v_j$ can be coloured with 'F'. A similar argument holds for axioms, which do not change the colour of the affected edge.

This line of reasoning leads to the following observation:

Lemma A.2.1. *A proof of inconsistency, which is a sub-graph of the congruence-closed $\mathcal{L}$-graph $\mathcal{G}(V, E)$ obtained using the algorithm in Section A.2, can be transformed into a colourable graph.*

Furthermore, given that an $\mathcal{L}$-graph $\mathcal{G}(V, E)$ represents a formula $F \wedge G$, which is a *conjunction* of atoms, the formula represented by a sub-graph is implied by $F \wedge G$. Thus, the proof of inconsistency is implied by the original formula $F \wedge G$.

It is straight forward to map a proof of inconsistency represented by a graph to a tree-shaped proof graph as described in Definition 3.5.1 in Section 3.5.1. Essentially, the tree-shaped proof $(V_P, E_P, \ell_P, \mathrm{s}_P)$ is the dual graph of the graph $\mathcal{G}(V, E)$ representing the

[5]Interpreted function symbols and constants are considered logical symbols.

proof of inconsistency. Each edge $(v_i \xrightarrow{\rhd} v_j) \in E$ maps to a vertex $v \in E_P$ such that $\ell(v) = (t_i \rhd t_j)$. Each congruence edge (Figure A.1(a)) maps to an application of the inference rule Cong in Table 3.7. Similarly, each derived edge (Figure A.1(b)) maps to the respective inference rule. The contradictory cycle corresponds to a final inference step deriving false.

Appendix B

Proofs

B.1 Proofs for Section 2.4

Lemma B.1.1. *The strongest postcondition distributes over disjunctions:*

$$sp(\texttt{stmt}, P_1 \vee P_2) = sp(\texttt{stmt}, P_1) \vee sp(\texttt{stmt}, P_2)$$

Conversely, the weakest (liberal) precondition distributes over conjunctions:

$$w(l)p(\texttt{stmt}, P_1 \wedge P_2) = w(l)p(\texttt{stmt}, P_1) \wedge w(l)p(\texttt{stmt}, P_2)$$

Proof. We consider each type of statement in Table 2.1 in Section 2.1 separately.

- $\texttt{x} := e$

$$\begin{aligned} sp(\texttt{x} := e, P_1 \vee P_2) &= \exists \texttt{x}' \,.\, \texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge (P_1 \vee P_2)[\texttt{x}/\texttt{x}'] \\ &= \exists \texttt{x}' \,.\, \texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge (P_1[\texttt{x}/\texttt{x}'] \vee P_2[\texttt{x}/\texttt{x}']) \\ &= \exists \texttt{x}' \,.\, (\texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge P_1[\texttt{x}/\texttt{x}']) \vee (\texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge P_2[\texttt{x}/\texttt{x}']) \\ &= \exists \texttt{x}' \,.\, (\texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge P_1[\texttt{x}/\texttt{x}']) \vee \exists \texttt{x}' \,.\, (\texttt{x} = e[\texttt{x}/\texttt{x}'] \wedge P_2[\texttt{x}/\texttt{x}']) \\ &= sp(\texttt{x} := e, P_1) \vee sp(\texttt{x} := e, P_2) \end{aligned}$$

$$
\begin{aligned}
w(l)p(\texttt{x} := e, P_1 \wedge P_2) &= (P_1 \wedge P_2)[\texttt{x}/e] \\
&= P_1[\texttt{x}/e] \wedge P_1[\texttt{x}/e] \\
&= w(l)p(\texttt{x} := e, P_1) \wedge w(l)p(\texttt{x} := e, P_2)
\end{aligned}
$$

- $\texttt{[}R\texttt{]}$

$$
sp([R], P_1 \vee P_2) = (R \wedge P_1) \vee (R \wedge P_2) = sp([R], P_1) \vee sp(R, P_2)
$$

$$
w(l)p([R], P_1 \wedge P_2) = (R \Rightarrow (P_1 \wedge P_2)) = \\
(R \Rightarrow P_1) \wedge (R \Rightarrow P_2) = w(l)p([R], P_1) \wedge w(l)p([R], P_2)
$$

- $\texttt{assert}(R)$

$$
sp(\texttt{assert}(R), P_1 \vee P_2) = (R \wedge P_1) \vee (R \wedge P_2) = \\
sp(\texttt{assert}(R), P_1) \vee sp(\texttt{assert}(R), P_2)
$$

$$
wp(\texttt{assert}(R), P_1 \wedge P_2) = (R \wedge P_1) \wedge (R \wedge P_2) = \\
wp(\texttt{assert}(R), P_1) \wedge wp(\texttt{assert}(R), P_2)
$$

$$
wlp(\texttt{assert}(R), P_1 \wedge P_2) = (R \Rightarrow (P_1 \wedge P_2)) = \\
(R \Rightarrow P_1) \wedge (R \Rightarrow P_2) = wlp(\texttt{assert}(R), P_1) \wedge wlp(\texttt{assert}(R), P_2)
$$

- $\texttt{stmt}_1;\ \texttt{stmt}_2$

$$
\begin{aligned}
sp(\texttt{stmt}_1;\ \texttt{stmt}_2, P_1 \vee P_2) &= sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P_1) \vee sp(\texttt{stmt}_1, P_2)) \\
&= sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P_1)) \vee sp(\texttt{stmt}_2, sp(\texttt{stmt}_1, P_2)) \\
&= sp(\texttt{stmt}_1; \texttt{stmt}_2, P_1) \vee sp(\texttt{stmt}_1; \texttt{stmt}_2, P_2)
\end{aligned}
$$

$$
\begin{aligned}
w(l)p(\texttt{stmt}_1;\ \texttt{stmt}_2, P_1 \wedge P_2) &= w(l)p(\texttt{stmt}_1, w(l)p(\texttt{stmt}_2, P_1) \wedge w(l)p(\texttt{stmt}_2, P_2)) \\
&= w(l)p(\texttt{stmt}_1, w(l)p(\texttt{stmt}_2, P_1)) \wedge \\
&\qquad w(l)p(\texttt{stmt}_1, w(l)p(\texttt{stmt}_2, P_2)) \\
&= w(l)p(\texttt{stmt}_1; \texttt{stmt}_2, P_1) \wedge w(l)p(\texttt{stmt}_1; \texttt{stmt}_2, P_2)
\end{aligned}
$$

The last case can be extended to arbitrary paths π length by means of induction over the length of π. □

B.2 Proofs for Section 3.3.3

Remark. It follows from Definitions 3.3.7 and 3.3.10 that $\ell(v)\downharpoonleft_{\perp,L} = \emptyset$ for all vertices v. Therefore, the following two equalities hold for any clause $C = \ell(v)$ in a resolution refutation R:

- $C\downharpoonleft_{\mathsf{b},L} = (C\upharpoonleft_{\mathsf{b},L} \setminus C\upharpoonleft_{\mathsf{a},L})$
- $C\downharpoonleft_{\mathsf{a},L} = (C\upharpoonleft_{\mathsf{a},L} \setminus C\upharpoonleft_{\mathsf{b},L})$

We make repeated use of these equalities in this section.

The proofs in this section owe their concise form to my collaborator Vijay D'Silva, who significantly improved my initial version. Also, Theorem 3.3.4 is a generalised version of his initial proof showing that McMillan's interpolation system is stronger than the interpolation system of Huang, Krajíček, and Pudlák.

Theorem 3.3.3 (Correctness of Labelled Interpolation Systems)**.** *For any resolution refutation R of a formula $A \wedge \overline{B}$ and locality preserving labelling function L, $\mathsf{Itp}(L, R, A, \overline{B})$ is a reverse interpolant for $(A, \overline{B})$.*

Proof. By induction over the structure of the refutation R for $A \wedge \overline{B}$. Let I be the partial interpolant at a vertex v labelled with a clause $C = \ell(v)$. We show that every such I and C satisfy the following conditions:

1. $A \wedge \neg(C\upharpoonleft_{\mathsf{a},L}) \Rightarrow I$,
2. $\overline{B} \wedge \neg(C\upharpoonleft_{\mathsf{b},L}) \Rightarrow \neg I$, and

3. $\text{Atoms}(I) \subseteq \text{Atoms}(A) \cap \text{Atoms}(\overline{B})$.

For the sink s_R with $\ell(s_R) = \square$, this establishes Theorem 3.3.3. The labelling function L, being unique in this proof, is omitted from subscripts.

Base case. Let v be an initial vertex and let $C = \ell_R(v)$.

1. $C \in A$:

   (a) $A \wedge \neg(C\restriction_{\mathsf{a}}) \Rightarrow C\downharpoonright_{\mathsf{b}}$, is equivalent to $A \Rightarrow (C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}} L) \vee C\restriction_{\mathsf{a}}$. This holds because $(C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}}) \vee C\restriction_{\mathsf{a}} = C$, and $A \Rightarrow C$ because $C \in A$.

   (b) $\overline{B} \wedge \neg(C\restriction_{\mathsf{b}}) \Rightarrow \neg(C\downharpoonright_{\mathsf{b}})$, is equivalent to $\overline{B} \wedge (C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}}) \Rightarrow C\restriction_{\mathsf{b}}$. This holds because $(C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}}) \subseteq C\restriction_{\mathsf{b}}$ and clauses represent disjunctions.

   (c) For all literals $l \in (C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}})$ the following conditions hold:

   - $\text{Atoms}(l) \subseteq \text{Atoms}(A)$, since $C \in A$.
   - $L(v, l) = \mathsf{b}$. Therefore, by Definition 3.3.8, $\text{Atoms}(l) \subseteq \text{Atoms}(\overline{B})$.

   This establishes that $\text{Atoms}(C\restriction_{\mathsf{b}} \setminus C\restriction_{\mathsf{a}}) \subseteq \text{Atoms}(A) \cap \text{Atoms}(\overline{B})$.

2. $C \in \overline{B}$: Symmetric to $C \in A$.

Induction step. We first prove a useful equality. Let v be an internal vertex of R, and $piv(v) = \mathtt{x}$. Observe that

$$(\ell(v^+) \setminus \{\mathtt{x}\})\restriction_{\mathsf{c}} \vee (\ell(v^-) \setminus \{\neg\mathtt{x}\})\restriction_{\mathsf{c}} = \ell(v)\restriction_{\mathsf{c}} \tag{B.1}$$

holds for a symbol $\mathsf{c} \in \{\mathsf{a}, \mathsf{b}\}$. This is because for any $l \in \ell(v^+)$, if $\text{Atoms}(l) \neq \{\mathtt{x}\}$, then $L(v^+, l) \sqsubseteq L(v, l)$. The same holds for $l \in \ell(v^-)$. Thus, if $\text{Atoms}(l) \neq \{\mathtt{x}\}$ and $l \in C\restriction_{\mathsf{c}}$, then $l \in \ell(v)\restriction_{\mathsf{c}}$. Conversely, if $l \in \ell(v)\restriction_{\mathsf{c}}$, then $\mathsf{c} \sqsubseteq L(v, l)$ by the definition of projection.

From Definition 3.3.8, $L(v,l) = L(v^+,l) \sqcup L(v^-,l)$, thus, if $\mathsf{c} \sqsubseteq L(v,l)$ and $\mathsf{c} \neq \mathsf{ab}$, then $\mathsf{c} \sqsubseteq L(v^+,l)$ or $\mathsf{c} \sqsubseteq L(v^-,l)$. It follows that $l \in (\ell(v^+) \setminus \{\mathtt{x}\})\restriction_{\mathsf{c}}$ or $l \in (\ell(v^-) \setminus \{\neg\mathtt{x}\})\restriction_{\mathsf{c}}$.

For the induction step, let $\ell(v^+) = \mathtt{x} \vee C$, $\ell(v^-) = \neg\mathtt{x} \vee D$ and $piv(v) = \mathtt{x}$. We perform a case split on $L(v^+,\mathtt{x}) \sqcup L(v^-,\neg\mathtt{x})$:

1. $L(v^+,\mathtt{x}) \sqcup L(v^-,\neg\mathtt{x}) = \mathsf{a}$:

   Induction hypothesis:

$$A \wedge \neg\mathtt{x} \wedge \neg(C\restriction_{\mathsf{a}}) \Rightarrow I_1 \text{ and } \quad \overline{B} \wedge \neg(C\restriction_{\mathsf{b}}) \Rightarrow \neg I_1 \text{ and}$$
$$A \wedge \mathtt{x} \wedge \neg(D\restriction_{\mathsf{a}}) \Rightarrow I_2 \text{ and } \quad \overline{B} \wedge \neg(D\restriction_{\mathsf{b}}) \Rightarrow \neg I_2\,.$$

   It follows that $A \wedge \mathtt{x} \Rightarrow I_1 \vee C\restriction_{\mathsf{a}}$ and $A \wedge \neg\mathtt{x} \Rightarrow I_2 \vee D\restriction_{\mathsf{a}}$,

$$\text{and that } A \wedge (\mathtt{x} \vee \neg\mathtt{x}) \Rightarrow I_1 \vee I_2 \vee \underbrace{C\restriction_{\mathsf{a}} \vee D\restriction_{\mathsf{a}}}_{(C\vee D)\restriction_{\mathsf{a}} \text{ by (B.1)}}, \qquad \text{by disjunction,}$$
$$\text{wherefore } A \wedge \neg((C \vee D)\restriction_{\mathsf{a}}) \Rightarrow I_1 \vee I_2 \qquad \text{as required.}$$

$$\text{Similarly, } \overline{B} \Rightarrow \neg I_1 \vee C\restriction_{\mathsf{b}} \text{ and } \overline{B} \Rightarrow \neg I_2 \vee D\restriction_{\mathsf{b}}\,.$$
$$\text{Thus, } \overline{B} \Rightarrow (\neg I_1 \wedge \neg I_2) \vee (C \vee D)\restriction_{\mathsf{b}} \qquad \text{by conjunction}$$
$$\text{hence, } \overline{B} \wedge \neg((C \vee D)\restriction_{\mathsf{b}}) \Rightarrow \neg(I_1 \vee I_2) \qquad \text{as required.}$$

   $\text{Atoms}(I_1 \vee I_2) \subseteq \text{Atoms}(A) \cap \text{Atoms}(\overline{B})$ holds because both $\text{Atoms}(I_1)$ and $\text{Atoms}(I_2)$ are contained in $\text{Atoms}(A) \cap \text{Atoms}(\overline{B})$.

2. $L(v^+,\mathtt{x}) \sqcup L(v^-,\neg\mathtt{x}) = \mathsf{b}$: The proof is symmetric to the first case.

3. $L(v^+,\mathtt{x}) \sqcup L(v^-,\neg\mathtt{x}) = \mathsf{ab}$:

Induction hypothesis for $L(v^+, \mathtt{x}) = \mathsf{a}$ *and* $L(v^-, \neg\mathtt{x}) = \mathsf{b}$*:*

$$A \wedge \neg\mathtt{x} \wedge \neg(C{\restriction}_{\mathsf{a}}) \Rightarrow I_1 \text{ and } \quad \overline{B} \wedge \quad \neg(C{\restriction}_{\mathsf{b}}) \Rightarrow \neg I_1 \text{ and}$$

$$A \wedge \quad \neg(D{\restriction}_{\mathsf{a}}) \Rightarrow I_2 \text{ and } \quad \overline{B} \wedge \mathtt{x} \wedge \neg(D{\restriction}_{\mathsf{b}}) \Rightarrow \neg I_2\,.$$

All cases in which $L(v^+, \mathtt{x}) \neq L(v^-, \neg\mathtt{x})$ are similar. The induction hypothesis in these cases can be extended to the induction hypothesis for $L(v^+, \mathtt{x}) = L(v^-, \neg\mathtt{x}) = \mathsf{ab}$ below:

$$A \wedge \neg\mathtt{x} \wedge \neg(C{\restriction}_{\mathsf{a}}) \Rightarrow I_1 \text{ and } \quad \overline{B} \wedge \neg\mathtt{x} \wedge \neg(C{\restriction}_{\mathsf{b}}) \Rightarrow \neg I_1 \text{ and}$$

$$A \wedge \mathtt{x} \wedge \neg(D{\restriction}_{\mathsf{a}}) \Rightarrow I_2 \text{ and } \quad \overline{B} \wedge \mathtt{x} \wedge \neg(D{\restriction}_{\mathsf{b}}) \Rightarrow \neg I_2\,.$$

We can infer that

$$A \wedge \neg(C{\restriction}_{\mathsf{a}}) \wedge \neg(D{\restriction}_{\mathsf{a}}) \Rightarrow (x \vee I_1) \wedge (\neg\mathtt{x} \vee I_2) \text{ and so}$$

$$A \wedge \neg((C \vee D){\restriction}_{\mathsf{a}} \Rightarrow (\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2).$$

Finally,

$$\overline{B} \wedge \neg(C{\restriction}_{\mathsf{b}}) \wedge \neg(D{\restriction}_{\mathsf{b}}) \Rightarrow (\mathtt{x} \vee \neg I_1) \wedge (\neg\mathtt{x} \vee \neg I_2) \text{ is equivalent to}$$

$$\overline{B} \wedge \neg(C \vee D){\restriction}_{\mathsf{b}} \Rightarrow \neg((\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2))\,.$$

Note that $\mathtt{x} \in \text{Atoms}(A) \cap \text{Atoms}(\overline{B})$ due to $L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{ab}$ and Definition 3.3.8, and therefore $\text{Atoms}((\mathtt{x} \vee I_1) \wedge (\neg\mathtt{x} \vee I_2)) \subseteq \text{Atoms}(A) \cap \text{Atoms}(\overline{B})$ holds.

□

Lemma 3.3.1. *Let* R *be a resolution refutation for the formula* $A \wedge \overline{B}$*. The labelling functions* L_{HKP}*,* L_M *and* $L_{M'}$ *are defined for initial vertices* v *and literals* $l \in \ell(v)$ *as follows:*

Atoms(l)	$L_M(v,l)$	$L_{HKP}(v,l)$	$L_{M'}(v,l)$
A-local	a	a	a
shared	b	ab	a
B-local	b	b	b

The following equalities hold for all vertices v in the proof R:

$$\begin{aligned}
\mathsf{Itp}_M(R, A, \overline{B})(v) &\equiv \mathsf{Itp}(L_M, R, A, \overline{B})(v) \\
\mathsf{Itp}_{HKP}(R, A, \overline{B})(v) &\equiv \mathsf{Itp}(L_{HKP}, R, A, \overline{B})(v) \\
\neg\mathsf{Itp}_M(R, \overline{B}, A)(v) &\equiv \mathsf{Itp}(L_{M'}, R, A, \overline{B})(v)
\end{aligned}$$

Proof. We prove the three equalities

$$\begin{aligned}
\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})(v) &\equiv \mathsf{Itp}(L_M, R, A, \overline{B})(v)\,, \\
\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(v) &\equiv \mathsf{Itp}(L_{HKP}, R, A, \overline{B})(v)\,, \text{and} \\
\neg\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)(v) &\equiv \mathsf{Itp}(L_{M'}, R, A, \overline{B})(v)
\end{aligned}$$

separately, by induction over the structure of R. For each vertex v in R with $\ell_R(v) = C$, let I_L^v be the annotation corresponding to $\mathsf{Itp}(L, R, A, \overline{B})(v)$ where $L \in \{L_M, L_{HKP}, L_{M'}\}$, and let I_M^v, I_{HKP}^v, and $I_{M'}^v$ be the annotations generated by $\mathsf{Itp}_{\mathrm{M}}(R, A, \overline{B})(v)$, $\mathsf{Itp}_{\mathrm{HKP}}(R, A, \overline{B})(v)$, and $\mathsf{Itp}_{\mathrm{M}}(R, \overline{B}, A)(v)$, respectively.

$\underline{\mathsf{Itp}_{\mathbf{HKP}}(R, A, \overline{B})(v) = \mathsf{Itp}(L_{HKP}, R, A, \overline{B})(v).}$

Base case. Let v be an initial vertex and let $C = \ell(v)$.

- If $C \in A$, then $I_L^v = C{\downharpoonleft}_{\mathsf{b}} = \bot = I_{HKP}^v$, since $C{\downharpoonleft}_{\mathsf{b}}$ is empty.
- If $C \in \overline{B}$, then $I_L^v = \neg(C{\downharpoonleft}_{\mathsf{a}}) = \top = I_{HKP}^v$, since $C{\downharpoonleft}_{\mathsf{a}}$ is empty.

Induction hypothesis: $I_L^{v^+} = I_{HKP}^{v^+}$ and $I_L^{v^-} = I_{HKP}^{v^-}$.

Induction step. Let v in R be an internal vertex and let $\mathtt{x} = piv(v)$.

1. If $\mathtt{x} \in \mathrm{Atoms}(A) \setminus \mathrm{Atoms}(\overline{B})$, then $L_{HKP}(v^+, \mathtt{x}) = L_{HKP}(v^-, \neg\mathtt{x}) = \mathsf{a}$, since $\mathtt{x}$ is A-local. It follows that $I_L^v = I_{HKP}^{v^+} \vee I_{HKP}^{v^-}$.
2. If $\mathtt{x} \in \mathrm{Atoms}(\overline{B}) \setminus \mathrm{Atoms}(A)$, then $L_{HKP}(v^+, \mathtt{x}) = L_{HKP}(v^-, \neg\mathtt{x}) = \mathsf{b}$, since $\mathtt{x}$ is B-local. It follows that $I_L^v = I_{HKP}^{v^+} \wedge I_{HKP}^{v^-} = I_{HKP}^v$.

3. If $\mathtt{x} \in \text{Atoms}(\overline{B}) \cap \text{Atoms}(A)$, then $L_{HKP}(v^+, x) = L_{HKP}(v^-, \overline{x}) = \mathsf{ab}$, because $\mathtt{x}$ and $\neg\mathtt{x}$ are have the label ab in all initial clauses. As ab is the top of the lattice of symbols, it follows that $I_L^v = (x \vee I_{HKP}^{v^+}) \wedge (\neg\mathtt{x} \vee I_{HKP}^{v^-}) = I_{HKP}^v$.

$\mathsf{Itp}_{\mathbf{M}}(R, A, \overline{B})(v) = \mathsf{Itp}(L_M, R, A, \overline{B})(v)$.

Base case. Let v in R be an initial vertex and let $\ell_R(v) = C$.

- If $C \in A$, then $I_L^v = C\lfloor_{\mathsf{b}} = C|_B = I_M^v$, because if $l \in C\lfloor_{\mathsf{b}}$ then $\text{Atoms}(l) \in \text{Atoms}(\overline{B})$.
- If $C \in \overline{B}$, then $I_L^v = \neg(C\lfloor_{\mathsf{a}}) = \top = I_M^v$, since $C\lfloor_{\mathsf{a}}$ is empty.

Induction hypothesis: $I_L^{v^+} = I_M^{v^+}$ and $I_L^{v^-} = I_M^{v^-}$.

Induction step. Let v in R be an internal vertex and let $\mathtt{x} = piv(v)$.

1. If $\mathtt{x} \in \text{Atoms}(A) \setminus \text{Atoms}(\overline{B})$, then $L_M(v^+, \mathtt{x}) = L_M(v^-, \neg\mathtt{x}) = \mathsf{a}$, because $\mathtt{x}$ is A-local and all literals over $\mathtt{x}$ are labelled a. Thus, $I_L^v = I_M^{v^+} \vee I_M^{v^-} = I_M^v$.
2. If $\mathtt{x} \in \text{Atoms}(\overline{B})$, then $L_M(v^+, \mathtt{x}) = L_M(v^-, \neg\mathtt{x}) = \mathsf{b}$, because all literals that are not A-local are labelled b. Thus, $I_L^v = I_M^{v^+} \wedge I_M^{v^-} = I_M^v$.

$\neg\mathsf{Itp}_M(R, \overline{B}, A)(v) = \mathsf{Itp}(L_{M'}, R, A, \overline{B})(v)$.

Base case. Let v in R be an initial vertex and let $\ell_R(v) = C$. Then

- If $C \in A$, then $I_L^v = C\lfloor_{\mathsf{b}} = \bot = \neg\top = \neg\mathsf{Itp}_M(R, \overline{B}, A)(v) = \neg I_{M'}^v$, since $C\lfloor_{\mathsf{b}}$ is empty.
- If $C \in \overline{B}$, then $I_L^v = \neg(C\lfloor_{\mathsf{a}}) = \neg(C|_A) = \neg\mathsf{Itp}_M(R, \overline{B}, A)(v) = \neg I_{M'}^v$.

Induction hypothesis: $I_L^{v^+} = \neg I_{M'}^{v^+}$ and $I_L^{v^-} = \neg I_{M'}^{v^-}$.

Induction step. Let v in R be an internal vertex and let $\mathtt{x} = piv(v)$.

1. If $\texttt{x} \in \text{Atoms}(A)$, then $L_{M'}(v^+, \texttt{x}) = L_{M'}(v^-, \neg\texttt{x}) = \mathsf{a}$, because literals that are not B-local are labelled a. It follows that

$$\begin{aligned} I_L^v = I_L^{v^+} \vee I_L^{v^-} &= \neg I_{M'}^{v^+} \vee \neg I_{M'}^{v^-} \\ &= \neg(\mathsf{Itp}_M(R, \overline{B}, A)(v^+) \wedge \mathsf{Itp}_M(R, \overline{B}, A)(v^-)) \,. \end{aligned}$$

   Since $L_{M'}(v, \texttt{x}) = L_{M'}(v, \neg\texttt{x}) = \mathsf{a}$ in the $(A, \overline{B})$-refutation R, we have that $\texttt{x} \notin \text{Atoms}(\overline{B}) \setminus \text{Atoms}(A)$. It follows that in $\mathsf{Itp}_M(R, \overline{B}, A)$

$$\neg\mathsf{Itp}_M(R, \overline{B}, A)(v) = \neg(\mathsf{Itp}_M(R, \overline{B}, A)(v^+) \wedge \mathsf{Itp}_M(R, \overline{B}, A)(v^-)) = \neg I_{M'}^v \,.$$

2. If $\texttt{x} \in \text{Atoms}(\overline{B}) \setminus \text{Atoms}(A)$, then $L_{M'}(v^+, \texttt{x}) = L_{M'}(v^+, \neg\texttt{x}) = \mathsf{b}$, because B-local variables are labelled b. Thus,

$$\begin{aligned} I_L^v = I_L^{v^+} \wedge I_L^{v^-} &= \neg I_{M'}^{v^+} \wedge \neg I_{M'}^{v^-} \\ &= \neg(I_{M'}^{v^+} \vee I_{M'}^{v^-}) \end{aligned}$$

   Since $\texttt{x} \in \text{Atoms}(\overline{B}) \setminus \text{Atoms}(A)$, it follows that in $\mathsf{Itp}_M(R, \overline{B}, A)$

$$\neg\mathsf{Itp}_M(R, \overline{B}, A)(v) = \neg(\mathsf{Itp}_M(R, \overline{B}, A)(v^+) \vee \mathsf{Itp}_M(R, \overline{B}, A)(v^-)) = \neg I_{M'}^v \,.$$

□

B.3 Proofs for Section 3.3.4

Lemma 3.3.2. *Let L and L' be labelling functions for a resolution refutation R for $A \wedge \overline{B}$. If $L(v, l) \preceq L'(v, l)$ for all initial vertices v and literals $l \in \ell(v)$, then $L \preceq L'$.*

Proof. We show that $L(v, l) \preceq L'(v, l)$ for all v in R by means of structural induction.

Base case. If $v \in V_R$ is an initial vertex, $L(v, l) \preceq L'(v, l)$ holds by definition.

Induction hypothesis: For an internal vertex v and its ancestors v^+ and v^- and a literal l it holds that

$$L(v^+, l) \preceq L'(v^+, l) \text{ and } L(v^-, l) \preceq L'(v^-, l).$$

Induction step. Let $v \in V_R$ be an internal vertex with the ancestors v^+ and v^-. Furthermore, let $\ell_R(v^+) = C \vee \mathtt{x}$, $\ell_R(v^-) = D \vee \neg\mathtt{x}$, where $\mathtt{x} = piv(v)$.

We consider two cases:

1. If $l \notin \ell(v)$, then $L(v, l) = L'(v, l) = \bot$.
2. If $l \in \ell(v)$, there are three cases:
   - If $L(v, l) = \mathsf{b}$, then $L(v, l) \preceq L'(v, l)$ because b is the infimum of $(\mathcal{S}, \preceq)$.
   - If $L(v, l) = \mathsf{ab}$ then $\mathsf{ab} \preceq L'(v, l)$. For assume that this does not hold. Then $L'(v, l)$ must be b, implying that $L'(v^+, l)$ and $L'(v^-, l)$ are both b (by the definition of $\sqcup$ and Definition 3.3.7). Using the induction hypothesis, we further conclude that $L(v^+, l) = L(v^-, l) = \mathsf{b}$, which contradicts $L(v, l) = \mathsf{ab}$.
   - If $L(v, l) = \mathsf{a}$ then, by the induction hypothesis, $L'(v^+, l)$ and $L'(v^-, l)$ are either a or $\bot$. In both cases, the lemma holds.

□

Theorem 3.3.4. *Let L and L' be labelling functions for an refutation R of the formula $A \wedge \overline{B}$. If $L \preceq L'$, then $\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L', R, A, \overline{B})(\mathrm{s}_R)$.*

Proof. By structural induction over R. For any vertex $v \in V_R$, let $C = \ell_R(v)$ and let I_v and I'_v represent the annotations $\mathsf{Itp}(L, R, A, \overline{B})(v)$ and $\mathsf{Itp}(L', R, A, \overline{B})(v)$, respectively. We show that $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$ for all vertices $v \in V_R$. This establishes $I_{\mathrm{s}_R} \Rightarrow I'_{\mathrm{s}_R}$, i.e., $\mathsf{Itp}(L, R, A, \overline{B})(\mathrm{s}_R) \Rightarrow \mathsf{Itp}(L', R, A, \overline{B})(\mathrm{s}_R)$.

Base case. Let v be an initial vertex and let $\ell_R(v) = C$.

1. If $C \in A$, then $I_v = C|_{\mathsf{b},L} = (C{\upharpoonright}_{\mathsf{b},L} \setminus C{\upharpoonright}_{\mathsf{a},L})$. From Definition 3.3.8, it follows that $C{\upharpoonright}_{\mathsf{b},L}$ and hence I_v is a subset of $(C|_A \cap C|_B)$. Thus, $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$.

2. If $C \in \overline{B}$, then $I'_v = \neg(C\downharpoonleft_{\mathsf{a},L'}) = \neg(C\upharpoonleft_{\mathsf{a},L'} \setminus C\upharpoonleft_{\mathsf{b},L'})$. Thus, $\neg I'_v = (C\upharpoonleft_{\mathsf{a},L'} \setminus C\upharpoonleft_{\mathsf{b},L'})$. From Definition 3.3.8, we have that $\neg I'_v \subseteq (C|_A \cap C|_B)$. It follows that, $\neg I'_v \Rightarrow \neg I_v \vee (C|_A \cap C|_B)$, which is equivalent to $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$.

Induction step. Let v be an internal vertex in R and let $\ell_R(v^+) = (C \vee \mathtt{x})$ and $\ell_R(v^-) = (D \vee \neg\mathtt{x})$, where $\mathtt{x} = piv(v)$. The partial annotations are indicated as before.

Induction hypothesis:

$$I_{v^+} \Rightarrow I'_{v^+} \vee (C|_A \cap C|_B) \text{ and } I_{v^-} \Rightarrow I'_{v^-} \vee (C|_A \cap C|_B)$$

Recall from the proof of Lemma 3.3.2, that if $L(v^+, x) \preceq L'(v^+, \mathtt{x})$ and $L(v^-, \neg\mathtt{x}) \preceq L'(v^-, \neg\mathtt{x})$, then,

$$(L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x})) \preceq (L'(v^+, \mathtt{x}) \sqcup L'(v^-, \neg\mathtt{x}))\,. \tag{B.2}$$

We proceed by performing a case split over $L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x})$. Let I and I' be the partial interpolants due to $\mathsf{Itp}(L, R, A, \overline{B})$ and $\mathsf{Itp}(L', R, A, \overline{B})$, respectively.

1. $L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{a}$: Then $I_v = (I_{v^+} \vee I_{v^-})$, and by applying the induction hypothesis we derive $(I_{v^+} \vee I_{v^-}) \Rightarrow (I'_{v^+} \vee I'_{v^-}) \vee (C|_A \cap C|_B)$. From (B.2) we conclude that $L'(v^+, \mathtt{x}) \sqcup L'(v^-, \neg\mathtt{x}) = \mathsf{a}$, and therefore $I'_v = (I'_{v^+} \vee I'_{v^-})$. It follows that $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$.

2. $L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{ab}$: Then $I_v = (\mathtt{x} \vee I_{v^+}) \wedge (\neg\mathtt{x} \vee I_{v^-})$, and by applying the induction hypothesis we derive

$$(x \vee I_{v^+}) \wedge (\overline{x} \vee I_{v^-}) \Rightarrow ((x \vee I'_{v^+}) \wedge (\overline{x} \vee I'_{v^-})) \vee (C|_A \cap C|_B)\,.$$

From equation (B.2), we have that $\mathsf{ab} \preceq L'(v^+, \mathtt{x}) \sqcup L'(v^-, \neg\mathtt{x})$, so I'_v is either $(I'_{v^+} \vee I'_{v^-})$ or $(\mathtt{x} \vee I'_{v^+}) \wedge (\neg\mathtt{x} \vee I'_{v^-})$. In either case, I'_v is implied by $(\mathtt{x} \vee I'_{v^+}) \wedge (\neg\mathtt{x} \vee I'_{v^-})$, and therefore $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$.

3. $L(v^+, \mathtt{x}) \sqcup L(v^-, \neg\mathtt{x}) = \mathsf{b}$: Then $I = (I_{v^+} \wedge I_{v^-})$, and by applying the induction hypothesis we derive $(I_{v^+} \wedge I_{v^-}) \Rightarrow (I'_{v^+} \wedge I'_{v^-}) \wedge (C|_A \cap C|_B)$. Since $\mathsf{b} \preceq L'(v^+, \mathtt{x}) \sqcup L'(v^-, \neg\mathtt{x})$, I' is either $(I'_{v^+} \wedge I'_{v^-})$, $(\mathtt{x} \vee I'_{v^+}) \wedge (\neg\mathtt{x} \vee I'_{v^-})$, or $(I_{v^+} \vee I_{v^-})$. In all cases, I' is implied by $(I'_{v^+} \wedge I'_{v^-})$, and thus $I_v \Rightarrow I'_v \vee (C|_A \cap C|_B)$.

□

Theorem 3.3.5. *Let R be a refutation of $A \wedge \overline{B}$ and $\mathbb{L}_R$ be the set of locality preserving labelling functions over R. The structure $(\mathbb{L}_R, \preceq, \Uparrow, \Downarrow)$ is a complete lattice with L_M as the least and $L_{M'}$ as the greatest element.*

Proof. We show that $\preceq$ is a partial order, that $\Uparrow$ is the least upper bound on $\mathbb{L}_R$ and that $\Downarrow$ is the greatest lower bound on $\mathbb{L}_R$.

$\preceq$ is a partial order: The relation $\preceq$ is a total order on $\mathcal{S}$, extended pointwise to vertices of R and their literals. The point-wise extension preserves reflexivity, anti-symmetry and transitivity.

$(L_1 \Uparrow L_2) \in \mathbb{L}_R$: Let l be a literal that $\text{Atoms}(l) \subseteq \text{Atoms}(A) \setminus \text{Atoms}(\overline{B})$ or $\text{Atoms}(l) \subseteq \text{Atoms}(\overline{B}) \setminus \text{Atoms}(A)$ and let v be an initial vertex. Then $L_1(v, l)$ and $L_2(v, l)$ are equal to $\max(L_1(v, l), L_2(v, l))$. Therefore, $L_1(v, l) \Uparrow L_2(v, l)$ satisfies the conditions in Definition 3.3.8.

$\Uparrow$ is the least upper bound: We show that for any L', if $L_1 \preceq L'$ and $L_2 \preceq L'$, then $(L_1 \Uparrow L_2) \preceq L'$. Consider an initial vertex v and $l \in \ell(v)$. If $L_1(v, l) \preceq L'(v, l)$ and $L_2(v, l) \preceq L'(v, l)$, then $\max(L_1(v, l), L_2(v, l)) \preceq L'(v, l)$. Recall that $(L_1 \Uparrow L_2)(v, l) = \max(L_1(v, t), L_2(v, l))$. The case for internal nodes follows from Lemma 3.3.2.

$\Downarrow$ is the greatest lower bound: This case is similar to the above.

The lattice $(\mathbb{L}_R, \preceq, \Uparrow, \Downarrow)$ is complete because it is finite.

Least and greatest elements: We show that L_M and $L_{M'}$ are the least and greatest elements of $\mathbb{L}_R$. For this, we show that if L is a locality preserving labelling function, then $L_M \preceq L$. Let v be an initial vertex and $l \in \ell(v)$. If $\text{Atoms}(l) \subseteq \text{Atoms}(A) \setminus \text{Atoms}(\overline{B})$, it follows from Definition 3.3.8 that $L(v,l) = \mathsf{a}$ and that $L_M(v,l) = \mathsf{a}$, thus, $L_M(v,l) \preceq L(v,l)$. If If $\text{Atoms}(l) \subseteq \text{Atoms}(\overline{B})$, from the definition of L_M, we have that $L_M(v,l) = \mathsf{b}$, so $L_M(v,l) \preceq L(v,l)$ because b is the least element of $(\mathcal{S}, \preceq)$. For an internal vertex v and $l \in \ell(v)$, $L_M(v,l) \preceq L(v,l)$ by Lemma 3.3.2. The case for $L_{M'}$ is similar with $\text{Atoms}(l) \subseteq \text{Atoms}(A)$ being the distinguishing case. □

B.4 Proofs for Section 3.3.5

For convenience, the proofs from Figure 3.15(a) are shown in Figure B.1 for reference. Furthermore, we write $\text{Res}(\ell(v^+), \ell(v^-), piv(v))$ to denote the resolvent $\ell(v)$ of the clauses $\ell(v^+)$, $\ell(v^-)$ with respect to the pivot $piv(v)$. Moreover, given a literal l, we write $\text{Atom}(l)$ to denote the single element in the set $\text{Atoms}(l)$.

Lemma 3.3.3. *Let R be a proof with vertices v_1, v_2, v_3, v, w and labels as in Figure 3.15(a). If $l_0 \notin \ell_R(v_3)$ and $l_1 \notin \ell_R(v_2)$, then $R[w \rightleftharpoons v]$ is a resolution proof.*

Proof. Let ℓ' be the labelling function for $R[w \rightleftharpoons v]$. We show that for every internal vertex u in $R[w \rightleftharpoons v]$, $\ell'(u) = \text{Res}(\ell'(u^+), \ell'(u^-), piv'(u))$.

There are two cases. If u is not a descendant of w or v, then $\ell(u) = \ell'(u)$ and $piv(u) = piv'(u)$. The clause and pivot labels of these vertices does not change so $\ell'(u)$ labels u with the resolvent of its parents.

We now show that $\ell'(w) = \text{Res}(\ell'(v_1), \ell'(v_2), piv'(w))$.

$$\begin{aligned}
\ell'(w) &= l_0 \vee C_1 \vee C_3 \\
&= ((l_0 \vee l_1 \vee C_1) \setminus \{l_1\}) \vee ((\neg l_1 \vee C_3) \setminus \{\neg l_1\}) \\
&\quad \text{because } l_1 \notin C_1 \text{ and } \neg l_1 \notin C_3 \qquad \text{(B.3)} \\
&= \text{Res}\left((l_0 \vee l_1 \vee C_1), (\neg l_1 \vee C_3), \text{Atom}(l_1)\right) \\
&= \text{Res}\left(\ell'(v_1), \ell'(v_2), piv'(w)\right)
\end{aligned}$$

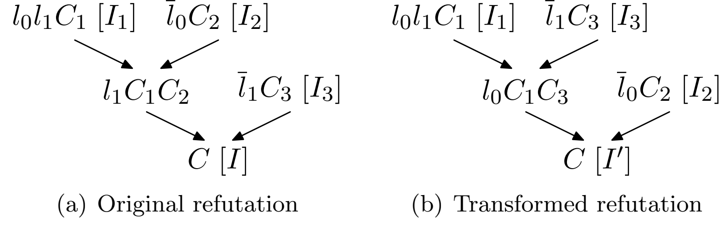


Figure B.1: Transformation of a proof by swapping resolution steps.

Next, we show that $\ell'(v) = \text{Res}(\ell'(w), \ell'(v_3), piv'(v))$.

$$\begin{aligned}
\ell'(v) &= \ell_R(v), \text{ from the definition of } R[w \rightleftharpoons v] \\
&= \text{Res}(\ell(v^+), \ell(v^-), piv(v)) \\
&= \text{Res}(\text{Res}(\ell_R(v_1), \ell_R(v_2), piv(w)), \ell(v_3), piv(v)) \\
&= \text{Res}(((l_0 \vee l_1 \vee C_1) \setminus \{l_0\}) \vee (\neg l_0 \vee C_2) \setminus \{\neg l_0\}, \neg l_1 \vee C_3, \text{Atom}(l_1)) \\
&= \ ((l_0 \vee l_1 \vee C_1) \setminus \{l_0\} \vee (\neg l_0 \vee C_2) \setminus \{\neg l_0\}) \setminus \{l_1\} \vee (\neg l_1 \vee C_3) \setminus \{\neg l_1\} \\
&= (l_0 \vee l_1 \vee C_1) \setminus \{l_0, l_1\} \vee (\neg l_0 \vee C_2) \setminus \{\neg l_0\} \vee (\neg l_1 \vee C_3) \setminus \{\neg l_1\} \\
&\quad \text{because } l_1 \notin C_2 \\
&= (l_0 \vee l_1 \vee C_1) \setminus \{l_0, l_1\} \vee (\neg l_0 \vee C_2) \setminus \{\neg l_0\} \vee (\neg l_1 \vee C_3) \setminus \{\neg l_1, l_0\} \\
&\quad \text{because } l_0 \notin C_3 \\
&= \ ((l_0 \vee l_1 \vee C_1) \setminus \{l_1\} \vee (\neg l_1 \vee C_3) \setminus \{\neg l_1\}) \setminus \{l_0\} \vee (\neg l_0 \vee C_2) \setminus \{\neg l_0\} \\
&= \text{Res}((l_0 \vee l_1 \vee C_1) \setminus \{l_1\} \vee (\neg l_1 \vee C_3) \setminus \{\neg l_1\}, \neg l_0 \vee C_2, \text{Atom}(l_0)) \\
&= \text{Res}(\text{Res}(l_0 \vee l_1 \vee C_1, \neg l_1 \vee C_3, \text{Atom}(l_1)), \neg l_0 \vee C_2, \text{Atom}(l_0)) \\
&= \text{Res}\left(\ell'(w), \ell'(v_3), piv'(v)\right)
\end{aligned} \tag{B.4}$$

□

Theorem 3.3.6. *Let R be a refutation of $A \wedge \overline{B}$, L a labelling function, w an internal vertex of R and (w, v) a merge-free edge. Let $\mathsf{c} = L(w^+, piv(w)) \sqcup L(w^-, \neg piv(w))$ and $\mathsf{d} = L(v^+, piv(v)) \sqcup L(v^-, \neg piv(v))$. Let Itp be a labelled interpolation system annotating vertices with partial interpolants I_1, I_2 and I_3 as in Figure 3.15(a).*

1. *If $\mathsf{c} \preceq \mathsf{d}$ and either $\mathsf{c} \neq \mathsf{d}$ or $\mathsf{c} \neq \mathsf{ab}$, $\mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v]) \Rightarrow \mathsf{Itp}(L, R)$.*

(a) Partial interpolants for Figure B.1(a) and Figure B.2(a)

$\mathsf{d} = L(v, l_1) \sqcup L(v, \neg l_1)$ \ $\mathsf{c} = L(w, l_0) \sqcup L(w, \neg l_0)$	a	ab	b
a	$(I_1 \vee I_2) \vee I_3$	$(l_0 \vee I_1) \wedge (\neg l_0 \vee I_2)$ $\vee I_3$	$(I_1 \wedge I_2) \vee I_3$
ab	$(l_1 \vee (I_1 \vee I_2))$ $\wedge(\neg l_1 \vee I_3)$	$(l_1 \vee ((l_0 \vee I_1) \wedge (\neg l_0 \vee I_2)))$ $\wedge(\neg l_1 \vee I_3)$	$(l_1 \vee (I_1 \wedge I_2))$ $\wedge(\neg l_1 \vee I_3)$
b	$(I_1 \vee I_2) \wedge I_3$	$(l_0 \vee I_1) \wedge (\neg l_0 \vee I_2)$ $\wedge I_3$	$(I_1 \wedge I_2) \wedge I_3$

(b) Partial interpolants for Figure B.1(b)

$L'(w, l_1) \sqcup L'(w, \neg l_1)$ \ $L'(v, l_0) \sqcup L'(v, \neg l_0)$	a	ab	b
a	$(I_1 \vee I_3) \vee I_2$	$(l_0 \vee (I_1 \vee I_3))$ $\wedge(\neg l_0 \vee I_2)$	$(I_1 \vee I_3) \wedge I_2$
ab	$(l_1 \vee I_1) \wedge (\neg l_1 \vee I_3)$ $\vee I_2$	$(l_0 \vee ((l_1 \vee I_1) \wedge (\neg l_1 \vee I_3)))$ $\wedge(\neg l_0 \vee I_2)$	$(l_1 \vee I_1) \wedge (\neg l_1 \vee I_3)$ $\wedge I_2$
b	$(I_1 \wedge I_3) \vee I_2$	$(l_0 \vee (I_1 \wedge I_3))$ $\wedge(\neg l_0 \vee I_2)$	$(I_1 \wedge I_3) \wedge I_2$

Table B.1: Interpolants for two subsequent resolution steps

2. In all other cases, if $I_2 \Rightarrow I_3$*, then* $\mathsf{Itp}(L[w \rightleftharpoons v], R[w \rightleftharpoons v]) \Rightarrow \mathsf{Itp}(L, R)$.

Proof. The two proofs R and $R[w \rightleftharpoons v]$ are shown in Figure B.1. We show that $I' \Rightarrow I$ for the different cases of the pair $\langle \mathsf{c}, \mathsf{d} \rangle$ in the theorem.

(1) $\mathsf{c} \preceq \mathsf{d} \wedge (\mathsf{c} \neq \mathsf{ab} \vee \mathsf{c} \neq \mathsf{d})$

- $\langle \mathsf{b}, \mathsf{b} \rangle$: $I' \Rightarrow I$ because I' is equivalent to I.
- $\langle \mathsf{b}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_1 \vee I_1) \wedge (\neg l_1 \vee I_3) \wedge I_2 \\ \Rightarrow & \quad (l_1 \vee I_1) \wedge (l_1 \vee I_2) \wedge (\neg l_1 \vee I_3) \\ \equiv & \quad (l_1 \vee (I_1 \wedge I_2)) \wedge (\neg l_1 \vee I_3) \\ \equiv & \quad I \end{aligned}$$

- $\langle \mathsf{b}, \mathsf{a} \rangle$:

$$\begin{aligned} I' \equiv\ & (I_1 \vee I_3) \wedge I_2 \\ \Rightarrow\ & (I_1 \wedge I_2) \vee I_3 \\ \equiv\ & I \end{aligned}$$

- $\langle \mathsf{a}, \mathsf{a} \rangle$: $I' \Rightarrow I$ because I' is equivalent to I.

- $\langle \mathsf{ab}, \mathsf{a} \rangle$:

$$\begin{aligned} I' \equiv\ & (l_0 \vee (I_1 \vee I_3)) \wedge (\neg l_0 \vee I_2) \\ \equiv\ & ((l_0 \vee I_1) \wedge (\neg l_0 \vee I_2)) \vee (I_3 \wedge (\neg l_0 \vee I_2)) \\ \Rightarrow\ & ((l_0 \vee I_1) \wedge (\neg l_0 \vee I_2)) \vee I_3 \\ \equiv\ & I \end{aligned}$$

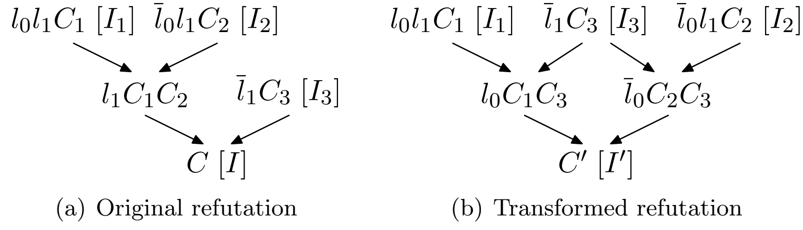
(2) $\neg(\mathbf{c} \preceq \mathbf{d}) \vee (\mathbf{c} = \mathbf{d} = \mathbf{ab})$ Assume that $I_2 \Rightarrow I_3$.

- $\langle \mathsf{ab}, \mathsf{b} \rangle$:

$$\begin{aligned} I' \equiv\ & (l_0 \vee (I_1 \wedge I_3)) \wedge (\neg l_0 \vee I_2) \\ \equiv\ & (l_0 \vee I_1) \wedge (l_0 \vee I_3) \wedge (\neg l_0 \vee I_2) \\ \overset{(I_2 \Rightarrow I_3)}{\Rightarrow}\ & (l_0 \vee I_1) \wedge (\neg l_0 \vee I_2) \wedge (l_0 \vee I_3) \wedge (\neg l_0 \vee I_3) \\ \equiv\ & (l_0 \vee I_1) \wedge (\neg l_0 \vee I_2) \wedge I_3 \wedge (l_0 \vee \neg l_0) \\ \equiv\ & (l_0 \vee I_1) \wedge (\neg l_0 \vee I_2) \wedge I_3 \equiv I \end{aligned}$$

- $\langle \mathsf{ab}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv\ & (l_0 \vee ((l_1 \vee I_1) \wedge (\neg l_1 \vee I_3))) \wedge (\neg l_0 \vee I_2) \\ \equiv\ & (l_0 \vee l_1 \vee I_1) \wedge (l_0 \vee \neg l_1 \vee I_3) \wedge (\neg l_0 \vee I_2) \wedge (l_1 \vee \neg l_1) \\ \equiv\ & (l_0 \vee l_1 \vee I_1) \wedge (l_0 \vee \neg l_1 \vee I_3) \wedge (\neg l_0 \vee l_1 \vee I_2) \wedge (\neg l_0 \vee \neg l_1 \vee I_2) \\ \overset{(I_2 \Rightarrow I_3)}{\Rightarrow}\ & (l_0 \vee l_1 \vee I_1) \wedge (\neg l_0 \vee l_1 \vee I_2) \wedge (l_0 \vee \neg l_1 \vee I_3) \wedge (\neg l_0 \vee \neg l_1 \vee I_3) \\ \equiv\ & (l_0 \vee l_1 \vee I_1) \wedge (\neg l_0 \vee l_1 \vee I_2) \wedge (\neg l_1 \vee I_3) \wedge (l_0 \vee \neg l_0) \\ \equiv\ & (l_1 \vee ((l_0 \vee I_1) \wedge (\neg l_0 \vee I_2))) \wedge (\neg l_1 \vee I_3) \\ \equiv\ & I \end{aligned}$$



(a) Original refutation (b) Transformed refutation

Figure B.2: Transformation of a refutation in which l_1 is a merge literal

$L(l_1) \sqcup L(\neg l_1)$ \ $L(l_0) \sqcup L(\neg l_0)$	a	ab	b
a	$I_1 \vee I_2 \vee I_3$	$(l_0 \vee I_1 \vee I_3)$ $\wedge(\neg l_0 \vee I_2 \vee I_3)$	$(I_1 \vee I_3) \wedge (I_2 \vee I_3)$
ab	$(l_1 \vee I_1) \wedge (\neg l_1 \vee I_3) \vee$ $(l_1 \vee I_2) \wedge (\neg l_1 \vee I_3)$	$(l_0 \vee ((l_1 \vee I_1) \wedge (\neg l_1 \vee I_3))) \wedge$ $(\neg l_0 \vee ((l_1 \vee I_2) \wedge (\neg l_1 \vee I_3)))$	$(l_1 \vee I_1) \vee (l_1 \wedge I_2)$ $\wedge(\neg l_1 \vee I_3)$
b	$(I_1 \wedge I_3) \vee (I_2 \wedge I_3)$	$(l_0 \vee (I_1 \wedge I_3))$ $\wedge(\neg l_0 \vee (I_2 \wedge I_3))$	$I_1 \wedge I_2 \wedge I_3$

Table B.2: Partial interpolant I' for different label values in Figure B.2(b)

- $\langle \mathsf{a}, \mathsf{b} \rangle$:

$$\begin{aligned} I' \equiv & \quad (I_1 \wedge I_3) \vee I_2 \\ \equiv & \quad (I_1 \vee I_2) \wedge (I_2 \vee I_3) \\ \overset{(I_2 \Rightarrow I_3)}{\Rightarrow} & \quad (I_1 \vee I_2) \wedge I_3 \\ \equiv & \quad I \end{aligned}$$

- $\langle \mathsf{a}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_1 \vee I_1) \wedge (\neg l_1 \vee I_3) \vee I_2 \\ \equiv & \quad (l_1 \vee I_1 \vee I_2) \wedge (\neg l_1 \vee I_2 \vee I_3) \\ \overset{(I_2 \Rightarrow I_3)}{\Rightarrow} & \quad (l_1 \vee (I_1 \vee I_2)) \wedge (\neg l_1 \vee I_3) \\ \equiv & \quad I \end{aligned}$$

□

The different cases of the proof are summarised in Table B.1.

Lemma B.4.1 shows that the transformation proposed by Jhala and McMillan [JM07] for the case that $l_1 \in C_2$ in Figure 3.15(a) (depicted in Figure B.2(a)) does not strengthen the resulting partial interpolant. Two cases of this lemma are shown in Figure B.3, where the partial interpolants correspond to a proof graph as in Figure B.2(a). In the first case, $L(v_1, l_0) \sqcup L(v_2, \neg l_0) = \mathsf{b}$ and $L(v_1, l_1) \sqcup L(v_2, \neg l_1) = \mathsf{a}$. In the second case, $L(v_1, l_0) \sqcup$

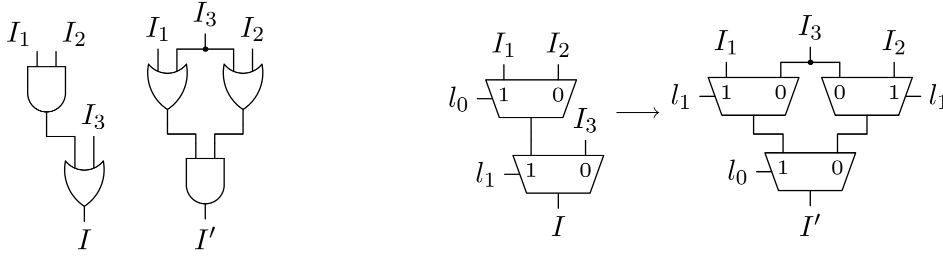


Figure B.3: Circuits representing the annotations I and I' in Figure B.2(a) and Figure B.2(b) for the cases $\langle \mathsf{c}, \mathsf{d} \rangle = \langle \mathsf{b}, \mathsf{a} \rangle$ and $\langle \mathsf{c}, \mathsf{d} \rangle = \langle \mathsf{ab}, \mathsf{ab} \rangle$.

$L(v_2, \neg l_0) = \mathsf{ab}$ and $L(v_1, l_1) \sqcup L(v_2, \neg l_1) = \mathsf{ab}$. One can verify that the circuits are logically equivalent. Though this transformation does not impact interpolant strength, it may be a useful intermediate step that enables other transformations.

Lemma B.4.1 (Redundant Transformation)**.** *Let R be a proof as illustrated in Figure B.2(a), and let R' the proof in Figure B.2(b). Furthermore, let I and I' be interpolants generated using either $\mathsf{Itp}(L_M, R, A, \overline{B})$, $\mathsf{Itp}(L_{HKP}, R, A, \overline{B})$, or $\mathsf{Itp}(L_{M'}, R, A, \overline{B})$. Then $I' \Leftrightarrow I$.*

Proof. Table B.2 summarises the possible partial interpolants at the vertex v for different labels of a literal. The labelling functions L_M, L_{HKP} and $L_{M'}$ assign the same label to all occurrences of a literal, hence we need not consider cases in which a literal has different labels in different clauses. For the proof, we show for each of these cases that the partial interpolant I' is logically equivalent to I.

We perform a case split on the tuple $\langle L(l_0) \sqcup L(\neg l_0), L(l_1) \sqcup L(\neg l_1) \rangle$:

- $\langle \mathsf{a}, \mathsf{a} \rangle$: holds trivially.
- $\langle \mathsf{ab}, \mathsf{a} \rangle$:

$$\begin{aligned} I' \equiv\ & (l_0 \vee I_1 \vee I_3) \wedge (\neg l_0 \vee I_2 \vee I_3) \\ \equiv\ & (l_0 \vee I_1) \wedge (\neg l_0 \vee I_2) \vee I_3 \ \equiv\ I \end{aligned}$$

- $\langle \mathsf{b}, \mathsf{a} \rangle$:

$$I' \equiv (I_1 \vee I_3) \wedge (I_2 \vee I_3) \equiv (I_1 \wedge I_2) \vee I_3 \equiv I$$

- $\langle \mathsf{a}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_1 \vee I_1) \wedge (\neg l_1 \vee I_3) \vee (l_1 \vee I_2) \wedge (\neg l_1 \vee I_3) \\ \equiv & \quad (l_1 \vee (I_1 \vee I_2)) \wedge (\neg l_1 \vee I_3) \;\equiv\; I \end{aligned}$$

- $\langle \mathsf{ab}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_0 \vee ((l_1 \vee I_1) \wedge (\neg l_1 \vee I_3))) \wedge (\neg l_0 \vee ((l_1 \vee I_2) \wedge (\neg l_1 \vee I_3))) \\ \equiv & \quad (l_0 \vee l_1 \vee I_1) \wedge (l_0 \vee \neg l_1 \vee I_3) \wedge (\neg l_0 \vee l_1 \vee I_2) \wedge (\neg l_o \vee \neg l_1 \vee I_3) \\ \equiv & \quad (l_0 \vee l_1 \vee I_1) \wedge (\neg l_0 \vee l_1 \vee I_2) \wedge (\neg l_1 \vee I_3) \wedge (l_0 \vee \neg l_0) \\ \equiv & \quad (l_1 \vee ((l_0 \vee I_1) \wedge (\neg l_0 \vee I_2))) \wedge (\neg l_1 \vee I_3) \;\equiv\; I \end{aligned}$$

- $\langle \mathsf{b}, \mathsf{ab} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_1 \vee I_1) \wedge (l_1 \vee I_2) \wedge (\neg l_1 \vee I_3) \\ \equiv & \quad (l_1 \vee (I_1 \wedge I_2)) \wedge (\neg l_1 \vee I_3) \;\equiv\; I \end{aligned}$$

- $\langle \mathsf{a}, \mathsf{b} \rangle$:

$$I' \;\equiv\; (I_1 \wedge I_3) \vee (I_2 \wedge I_3) \;\equiv\; (I_1 \vee I_2) \wedge I_3 \;\equiv\; I$$

- $\langle \mathsf{ab}, \mathsf{b} \rangle$:

$$\begin{aligned} I' \equiv & \quad (l_0 \vee (I_1 \wedge I_3)) \wedge (\neg l_0 \vee (I_2 \wedge I_3)) \\ \equiv & \quad (l_0 \vee I_1) \wedge (\neg l_0 \vee I_2) \wedge I_3 \;\equiv\; I \end{aligned}$$

- $\langle \mathsf{b}, \mathsf{b} \rangle$: Trivial.

□

B.5 Proofs for Section 3.5.2

Theorem 3.5.2. *Let $P = (V_P, E_P, \ell_P, \mathrm{s}_P)$ be a local refutation of the conjunction of the pair of formulae $(A, \overline{B})$. Then $\mathsf{Itp}^{\mathcal{T}}_{KV}(P, A, \overline{B})(v) \Rightarrow \mathsf{Itp}^{\mathcal{T}}_{KW}(P, A, \overline{B})(v)$ holds for each vertex $v \in V_P$ for which $\mathrm{Sym}(\ell(v)) \subseteq \mathrm{Sym}(A) \cap \mathrm{Sym}(B)$ holds.*

Proof. We prove Theorem 3.5.2 by induction over the structure of P. The base case is trivial, since $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})$ and $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v)$ assign the same partial interpolants to

the initial nodes of P.

Induction hypothesis: Let $v \in V_P$ be a vertex in P. Suppose that v is B-coloured (A-coloured, respectively). Let $\{C_1, \ldots, C_n\}$ denote the set A-premise(v) (or B-premise(v), respectively). Furthermore, let $I_1, \ldots, I_n$ and $I'_1, \ldots, I'_n$ be the corresponding partial interpolants generated by $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})$ and $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})$, respectively. Then it holds for all $i \in \{1..n\}$ that $I_i \Rightarrow I'_i$.

Induction step. For an internal node v with $C = \ell(v)$, we need to distinguish the following cases:

1. C is A-coloured. Then

$$\begin{aligned}\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})(v) &= \bigwedge_{i=1}^{n}(C_i \vee I_i) \wedge \neg \bigwedge_{i=1}^{n} C_i \quad \text{and} \\ \mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v) &= \bigvee_{i=1}^{n}(\neg C_i \wedge I'_i)\ .\end{aligned}$$

   We reason as follows:

$$\begin{aligned}\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})(v) &= \bigwedge_{j=1}^{n}(C_j \vee I_j) \wedge \bigvee_{i=1}^{n}(\neg C_i) \\ &= \bigvee_{i=1}^{n}\left((\neg C_i) \wedge \bigwedge_{j=1}^{n}(C_j \vee I_j)\right) \\ &= \bigvee_{i=1}^{n}\bigwedge_{j=1}^{n}((\neg C_i \wedge C_j) \vee (\neg C_i \wedge I_j)) \\ &= \bigvee_{i=1}^{n}\left((\neg C_i \wedge I_i) \wedge \bigwedge_{j=1, i\neq j}^{n}(\neg C_i \wedge C_j) \vee (\neg C_i \wedge I_j)\right) \\ &\Rightarrow \bigvee_{i=1}^{n}(\neg C_i \wedge I_i) \\ &\Rightarrow \bigvee_{i=1}^{n}(\neg C_i \wedge I'_i) \quad \text{(by the induction hypothesis)}\end{aligned}$$

2. C is B-coloured. Then

$$\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})(v) = \bigwedge_{i=1}^{n}(C_i \vee I_i) \quad \text{and}$$
$$\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v) = \bigvee_{i=1}^{n}(\neg C_i \wedge I'_i) \vee \bigwedge_{i=1}^{n} C_i \ .$$

We reason as follows:

$$\begin{aligned}
\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v) &= \bigwedge_{i=1}^{n}\left(C_i \vee \bigvee_{j=1}^{n}(\neg C_j \wedge I'_j)\right) \\
&= \bigwedge_{i=1}^{n}\bigvee_{j=1}^{n}\big((C_i \vee \neg C_j) \wedge (C_i \vee I'_j)\big) \\
&= \bigwedge_{i=1}^{n}\left((C_i \vee I'_i) \vee \bigvee_{j=1, i\neq j}^{n}(C_i \vee \neg C_j) \wedge (C_i \vee I'_j)\right)
\end{aligned}$$

Since the induction hypothesis warrants that $\bigwedge_{i=1}^{n}(C_i \vee I_i) \Rightarrow \bigwedge_{i=1}^{n}(C_i \vee I'_i)$ holds, we conclude that $\mathsf{Itp}^{\mathcal{T}}_{\mathrm{KV}}(P, A, \overline{B})(v) \Rightarrow \mathsf{Itp}^{\mathcal{T}}_{\mathrm{KW}}(P, A, \overline{B})(v)$.

□

Bibliography

[ABC+07] Alessandro Armando, Massimo Benerecetti, Dario Carotenuto, Jacopo Mantovani, and Pasquale Spica. The EUREKA tool for software model checking. In *Automated Software Engineering (ASE)*, pages 541–542. ACM, 2007.

[ABM06] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Model checking linear programs with arrays. In *Software Model Checking (SoftMC)*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 79–94. Elsevier, 2006.

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *European Conference on Planning (ECP)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.

[ACM04] Alessandro Armando, Claudio Castellini, and Jacopo Mantovani. Software model checking using linear constraints. In *International Conference on Formal Engineering Methods (IFCEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2004.

[And68] Peter B. Andrews. Resolution with merging. *Journal of the ACM*, 15(3):367–381, 1968.

[AQRX04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In *Concurrency Theory (CONCUR)*, pages 1–15. Springer, August 2004.

[Bal05] Thomas Ball. *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series II: Mathematics, Physics and Chemistry*, chapter Formalizing Counterexample-Driven Refinement with Weakest Preconditions, pages 121–139. Springer, 2005.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002. Formal Methods for Industrial Critical Systems (FMICS).

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *European Conference on Computer Systems (EuroSys)*, pages 73–85. ACM, 2006.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCDR04] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[BCF+06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 527–541. Springer, 2006.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.

[BCG+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 25–32. IEEE, 2009.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Verification (IFM)*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science (LICS)*, pages 428–439. IEEE, 1990.

[BD02] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 741–746. IEEE, 2002.

[BFLP03] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: Fast acceleration of symbolic transition systems. In *Computer Aided Verification (CAV)*, volume 2752 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BGK07] Nicolas Blanc, Alex Groce, and Daniel Kroening. Verifying C++ with STL containers via predicate abstraction. In *Automated Software Engineering (ASE)*, pages 521–524. IEEE, 2007.

[BHMR07a] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[BHMR07b] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Programming Language Design and Implementation (PLDI)*, pages 300–309. ACM, 2007.

[Bie08] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2–4):75–97, 2008.

[BIFH+09] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In *Haifa Verification Conference (HVC)*, volume 5394 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2009.

[BJ89] George Boolos and Richard C. Jeffrey. *Computability and logic.* Cambridge University Press, 3rd ed. edition, 1989.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[BKRW10] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Computer Science. Springer, 2010. to appear.

[BKS07] Thomas Ball, Orna Kupferman, and Mooly Sagiv. Leaping loops in the presence of abstraction. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2007.

[BKW07a] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. A complete bounded model checking algorithm for pushdown systems. In *Haifa Verification Conference (HVC)*, volume 4899 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2007.

[BKW07b] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based summarization for Boolean programs. In *Model Checking and Software Verification (SPIN)*, number 4595 in Lecture Notes in Computer Science, pages 131–148, 2007.

[BKWW08] Armin Biere, Daniel Kroening, Christoph M. Wintersteiger, and Georg Weissenbacher. *Digitaltechnik: Eine praxisnahe Einführung.* Springer, 2008.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213. ACM, 2001.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2002.

[BPR03] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. *Software Tools for Technology Transfer (STTT)*, 5(1):49–58, 2003.

[BR00a] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[BR00b] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking and Software Verification (SPIN)*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[BR02a] Thomas Ball and Sriram Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[BR02b] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3. ACM, 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BSV93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993.

[Bus99] Samuel R. Buss. Propositional proof complexity: An introduction. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 127–178. Springer, 1999.

[BZM08] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 304–308. Springer, 2008.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282. ACM, 1979.

[CCG+04] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGS09] Allesandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Interpolant generation for UTVPI⋆. In *Conference on Automated Deduction (CADE)*. Springer, 2009.

[CGS10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic*, 2010. to appear.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176. Springer, 2004.

[CKS06] Byron Cook, Daniel Kroening, and Natasha Sharygina. Over-approximating Boolean programs with unbounded thread creation. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 53–59. IEEE, 2006.

[CKSY04] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, 2004.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[Cla77] Edmund M. Clarke. Program invariants as fixed points. In *18th Annual Symposium on the Foundations of Computer Science*, pages 18–29. IEEE, November 1977.

[Cou00] Patrick Cousot. Partial completeness of abstract fixpoint checking. In *International Symposium on Abstraction, Reformulation, and Approximation (SARA)*, volume 1864 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction-refinement for termination. In *Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[Cra57a] William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[Cra57b] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CV03] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[D+82] Edsger W. Dijkstra et al. From predicate transformers to predicates, April 1982. Tuesday Afternoon Club Manuscript EWD821.

[DdM06] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DKPW09] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. Technical Report 652, Institute for Computer Science, ETH Zurich, November 2009.

[DKR10] Alastair Donaldson, Daniel Kroening, and Philipp Ruemmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DPWK10] Vijay D'Silva, Mitra Purandare, Georg Weissenbacher, and Daniel Kroening. Interpolant strength. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.

[D'S10] Vijay D'Silva. Propositional interpolation and abstract interpretation. In *Proceedings of the European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*. Springer, 2010.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2006.

[EKS08] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 5:27–56, June 2008. Special Issue on Constraints to Formal Verification.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for

dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919, pages 502–518. Springer, 2004.

[FGG+09] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstić, and Cesare Tinelli. Ground interpolation for the theory of equality. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5005 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2009.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, 2003.

[FL02] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, Lecture Notes in Computer Science, pages 145–156. Springer, 2002.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: A foundation for computer science*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[GKT09] Amit Goel, Sava Krstić, and Cesare Tinelli. Ground interpolation for combined theories. In *Conference on Automated Deduction (CADE)*. Springer, 2009.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Part II, vol. I. Technical Report 1, Institute for Advanced Studies, Princeton, N.J., April 1947.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[Heh84] Eric C. R. Hehner. Predicative programming Part I. *Communications of the ACM*, 27(2):134–143, 1984.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *Principles of Programming Languages (POPL)*, pages 471–482. ACM, 2010.

[HJM+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code.

In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer, 2002.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.

[HJMQ03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, 1976.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hua95] Guoxiang Huang. Constructing Craig interpolation formulas. In *Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 181–190. Springer, 1995.

[IYG+05] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-Soft: Software verification platform. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2005.

[IYG+08] Franjo Ivani, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[JCG09] Himanshu Jain, Edmund M. Clarke, and Orna Grumberg. Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations. *Formal Methods in System Design (FMSD)*, 35(1):6–39, 2009.

[Jha04] Ranjit Jhala. *Program Verification by Lazy Abstraction*. PhD thesis, University of California Berkeley, 2004.

[JIG+06] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2006.

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674. Springer, 2009.

[JM05] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47. ACM, 2005.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. *Logical Methods in Computer Science (LMCS)*, 3(4), 2007.

[KF70] James C. King and Robert W. Floyd. An interpretation oriented theorem prover over integers. In *ACM Symposium on Theory of Computing*, pages 169–179. ACM, 1970.

[KHCL07] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Automated Software Engineering (ASE)*, pages 389–392. ACM, 2007.

[Kin70] James C. King. *A program verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

[KMZ06] Deepak Kapur, Rupak Majumdar, and Calogero G. Zarba. Interpolation for data structures. In *Foundations of Software Engineering (FSE)*, pages 105–116. ACM, 2006.

[Kra97] Jan Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, 1997.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: An algorithmic point of view*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[Kuh62] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[KV09a] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2009.

[KV09b] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2009.

[KW06] Daniel Kroening and Georg Weissenbacher. Counterexamples with loops for predicate abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2006.

[KW07] Daniel Kroening and Georg Weissenbacher. Lifting propositional interpolants to the word-level. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 85–89. IEEE, 2007.

[KW09a] Daniel Kroening and Georg Weissenbacher. An interpolating decision procedure for transitive relations with uninterpreted functions. In *Haifa Verification Conference (HVC)*, Lecture Notes in Computer Science. Springer, 2009. To appear.

[KW09b] Daniel Kroening and Georg Weissenbacher. Model checking: Bugs in C-Programmen finden. *iX Magazin für professionelle Informationstechnik*, 5:159–162, May 2009.

[KW10] Daniel Kroening and Georg Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Aspects of Computing*, 22(2):105–128, March 2010.

[LL05] K. Rustan M. Leino and Franceso Logozzo. Loop invariants on demand. In *Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.

[Mae61] Shôji Maehara. On the interpolation theorem of Craig (in Japanese). *Sûgaku*, 12:235–237, 1961.

[McM92] McMillan, Kenneth L. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM05] Kenneth L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 2 edition, 1992.

[MS05] Orly Meir and Ofer Strichman. Yet another decision procedure for equality logic. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2005.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, 1989.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NO05] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NR10] Aditya V. Nori and Sriram K. Rajamani. An empirical study of optimizations in yogi. In *International Conference on Software Engineering (ICSE)*. ACM, May 2010. to appear.

[Pop34] Karl R. Popper. *Logik der Forschung*. Springer, 1934.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–25. Springer, 2004.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351, 1982.

[RSS07] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, pages 346–362. Springer, 2007.

[Sch02] Stefan Schwoon. *Model-checking pushdown systems*. PhD thesis, Technische Universitt München, 2002.

[SDGC07] Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 3–12. IEEE, 2007.

[SP81] M. Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tse83] G. Tseitin. On the complexity of proofs in poropositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983. Originally published 1970.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

[vEBG04] Robert A. van Engelen, Johnnie Birch, and Kyle A. Gallivan. Array data dependence testing with the chains of recurrences algebra. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)*, pages 70–81. IEEE, 2004.

[WBKW07] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *Automated Software Engineering (ASE)*, pages 501–504. IEEE, 2007.

[Weg60] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, 1960.

[Wei81] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE, 1981.

[WGI07] Chao Wang, Aarti Gupta, and Franjo Ivančić. Induction in CEGAR for detecting counterexamples. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 77–84. IEEE, 2007.

[YM05] Greta Yorsh and Madanlal Musuvathi. A combination method for generating interpolants. In *Conference on Automated Deduction (CADE)*, volume 3632 of *Lecture Notes in Computer Science*, pages 353–368, 2005.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design Automation and Test in Europe (DATE)*, page 10880. IEEE, 2003.