Dissertation

Satisfiability-Based Methods for Controller Synthesis

Robert Könighofer[1]

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
A-8010 Graz, Austria

Supervisor/First reviewer:	Prof. Roderick Bloem
Second reviewer:	Prof. Armin Biere

Graz, September 2015

[1] E-mail: robert.koenighofer@iaik.tugraz.at

Dissertation

SAT-Basierte Methoden zur Synthese von Controllern

Robert Könighofer[1]

Institut für Angewandte Informationsverarbeitung und
Kommunikationstechnologie (IAIK)
Technische Universität Graz
A-8010 Graz, Österreich

Betreuer/1. Gutachter:	Prof. Roderick Bloem
2. Gutachter:	Prof. Armin Biere

Graz, September 2015

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: robert.koenighofer@iaik.tugraz.at

Abstract

Synthesis is an appealing approach to construct hardware or software programs: a correct implementation is computed automatically from a declarative specification. *Controller synthesis* is a variant where (most) parts of the implementation are given, and only certain signals need to be synthesized. This allows for a mix of imperative and declarative programming, often referred to as *program sketching*, but also other applications such as automatic program repair. This thesis focuses on efficient controller synthesis methods for both hardware and software using decision procedures for the satisfiability of formulas.

In the hardware context, we focus on safety specifications. Existing synthesis approaches mostly use Binary Decision Diagrams (BDDs) as reasoning engine. In contrast, we present a number of novel algorithms that use decision procedures for propositional formulas (SAT solvers), Quantified Boolean Formulas (QBF solvers), or solvers for Effectively Propositional Logic (EPR). Our synthesis approach consists of two steps. The first step is to compute a strategy to satisfy the specification, where we present algorithms based on query learning, templates, reduction to EPR, and a parallelization that combines different methods. The second step is to compute a circuit implementing the strategy. Here, we present methods based on QBF certification, interpolation, query learning, and a parallelization as well. Our methods are augmented with numerous optimizations, including heuristics to expand quantifiers and to utilize unreachable states and variable independencies, down to low-level optimizations in the formula encoding. In an extensive experimental evaluation, we compare our algorithms and the effect of optimizations. We demonstrate that our satisfiability-based approach outperforms a simple BDD-based implementation and is even competitive with a highly optimized BDD-based tool. For specific benchmark classes, our techniques are particularly superior. Moreover, the circuits produced by our approach are smaller by more than one order of magnitude on average in our experiments. These excellent results are rooted in our optimizations and the careful utilization of solver features.

For software controller synthesis, we focus on the application of automatic program repair using assertions in the code as specification. Our approach consists of three steps. First, we perform program analysis using symbolic or concolic execution to lift the repair problem into the domain of logic. The second step is fault localization based on Model-Based Diagnosis and Satisfiability Modulo Theories (SMT) solving to identify potentially incorrect program parts. We also present an alternative approach that uses pre- and postconditions with deductive verification for fault localization. The third and central step is to synthesize replacements for the faulty program parts such that the specification is fulfilled. Our basic solution uses templates for new implementations and Counterexample-Guided Inductive Synthesis (CEGIS), enriched with heuristics, to search for suitable template instantiations with SMT solving. An improved variant interleaves the repair synthesis with on-the-fly program analysis to obtain more focused information about the program behavior. Our approach is designed to produce fine-grained and readable repairs and provides many parameters to trade accuracy for efficiency. Our proof-of-concept implementation operates on C programs. We present experimental results demonstrating that our approach can provide helpful diagnostic information in reasonable time. A comparison with existing tools indicates a higher diagnostic resolution in fault localization and better scalability in repair synthesis.

In summary, this thesis contributes towards scalability in synthesis with novel satisfiability-based algorithms and optimizations, and to its applicability in the interesting field of software program repair.

Keywords: Reactive Synthesis, Decision Procedures, Program Repair, Fault Localization.

Kurzfassung

Synthese ist ein attraktiver Ansatz um Hardware oder Software zu erstellen: ausgehend von einer deklarativen Spezifikation wird eine korrekte Implementierung automatisch berechnet. Die Synthese von Controllern ist eine Variante wo Teile der Implementierung bereits gegeben sind. Dies ermöglicht interessante Anwendungen wie Program-Sketching oder die automatische Reparatur von Programmen. Diese Dissertation beschäftigt sich mit der Synthese von Controllern, sowohl für Hardware als auch für Software, mittels Entscheidungsprozeduren für die Erfüllbarkeit von Formeln.

Im Kontext von Hardware fokussieren wir uns auf Safety-Spezifikationen. Existierende Ansätze verwenden zumeist Binäre Entscheidungsdiagramme (BDDs). Wir präsentieren hingegen neue Algorithmen basierend auf Entscheidungsprozeduren für Aussagenlogik (SAT-Solvern), quantifizierten booleschen Formeln (QBF-Solvern) oder Solvern für Effektively Propositional Logic (EPR). Unser Syntheseansatz besteht aus zwei Schritten. Der erste Schritt ist die Berechnung einer Strategie um die Spezifikation zu erfüllen. Hier präsentieren wir Algorithmen basierend auf Aktivem Lernen, Schablonen, Reduktion auf EPR, sowie eine Parallelisierung die mehrere Methoden kombiniert. Der zweite Schritt ist die Berechnung einer Schaltung. Hier präsentieren wir Methoden basierend auf QBF-Zertifizierung, Interpolation, Lernen und ebenfalls eine Parallelisierung. Unsere Methoden werden durch zahlreiche Optimierungen ergänzt. Dies inkludiert das Expandieren von Quantoren, das Ausnutzen von unerreichbaren Zuständen und von Unabhängigkeiten zwischen Variablen sowie Optimierungen in der Formelkodierung. In umfangreichen Experimenten vergleichen wir Algorithmen und studieren den Effekt von Optimierungen. Wir zeigen, dass unser Ansatz eine einfache BDD-basierte Lösung übertrifft und sogar auf Augenhöhe mit einem hochoptimierten Synthesetool steht. Die Schaltungen die unser Ansatz produziert sind in unseren Experimenten im Schnitt um mehr als eine Zehnerpotenz kleiner. Diese exzellenten Ergebnisse begründen sich in unseren Optimierungen und der sorgfältigen Verwendung von Solver-Funktionalitäten.

In der Synthese von Software-Controllern fokussieren wir uns auf die automatische Reparatur von Programmen mit Assertions als Spezifikation. Unser Ansatz umfasst drei Schritte. Der erste Schritt ist die Programmanalyse mittels symbolischer oder concolischer Ausführung um das Reparaturproblem in die Welt der Logik zu transformieren. Der zweite Schritt ist die Fehlerlokalisierung basierend auf modellbasierter Diagnose und Satisfiability Modulo Theories (SMT) Solving. Wir präsentieren auch einen alternativen Ansatz der Vor- und Nachbedingungen sowie deduktive Verifikation verwendet. Der dritte Schritt ist die Synthese von neuen Implementierungen der fehlerhaften Programmteile. Hier verwenden wir Schablonen und Counterexample-Guided Inductive Synthesis (CEGIS), realisiert mit SMT-Solving und erweitert mit Heuristiken. Eine verbesserte Variante verschränkt die Synthese von Reparaturen mit Programmanalyse nach Bedarf um fokussiertere Informationen über das Programmverhalten zu gewinnen. Unser Ansatz produziert feingranulare und lesbare Reparaturen und bietet viele Parameter um Genauigkeit gegen Effizienz einzutauschen. Unsere Prototyp-Implementierung arbeitet mit C Programmen. Wir präsentieren experimentelle Ergebnisse, die demonstrieren dass unser Ansatz hilfreiche diagnostische Information in vernünftiger Zeit liefern kann. Ein Vergleich mit existierenden Tools deutet auf bessere Genauigkeit in der Fehlerlokalisierung und bessere Skalierbarkeit in der Synthese von Reparaturen hin.

Zusammenfassend trägt diese Dissertation zur Skalierbarkeit von Synthese mit neuen SAT-basierten Algorithmen sowie zur Anwendbarkeit im Gebiet der automatischen Software-Reparatur bei.

Schlagworte: Reaktive Synthese, Entscheidungsprozeduren, Programmreparatur, Fehlerlokalisierung

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

. . . .

place, date (signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

. . . .

Ort, Datum (Unterschrift)

Acknowledgements

I am indebted to numerous people who supported me in my doctoral studies and this thesis.

First of all, I thank my supervisor Roderick Bloem for sparking my interest in research and formal methods, for guiding my work with his expertise and farsightedness, and for keeping me motivated. I could not have hoped for a better advisor.

Furthermore, I am grateful to Armin Biere for reviewing this thesis.

I also thank all my colleagues and co-authors. Particular thanks go to Georg Hofferek and Karin Greimel for mentoring me, especially in the beginning of my studies, to Martina Seidl, Florian Lonsing and Uwe Egly for introducing me to the exciting world of QBF and SAT, to the FoREnSiC team for the very productive collaboration, to Philipp Pani and Patrick Klampfl for contributing to the implementation of FoREnSiC and Demiurge, and to all colleagues from my working group for the countless discussions and fruitful collaborations.

Moreover, I thank my love Elisabeth for her support and understanding, and for being my source of energy. Last but not least, I thank my parents Rosa and Heinz and my whole family for keeping me grounded.

The research in this thesis has been financially supported by the European Commission through the projects DIAMOND (FP7-2009-IST-4-248613) and STANCE (317753), and by the Austrian Science Fund (FWF) through the research network RiSE (S11406-N23).

Robert Könighofer
Graz, Austria, September 2015

Danksagung

Ich bin vielen Menschen, die mich in der Erstellung dieser Dissertation unterstützt haben, zu großem Dank verpflichtet.

Zunächst danke ich meinem Betreuer Roderick Bloem dafür, dass er mein Interesse an der Forschung und an formalen Methoden geweckt hat, dass er meine Arbeit mit seiner Expertise und Weitsicht gelenkt hat und dafür, dass er mich immer motiviert hat. Ich hätte mir keinen besseren Betreuer vorstellen können.

Weiters danke ich Armin Biere für die Begutachtung dieser Dissertation.

Ich danke auch meinen Kollegen und Mitautoren. Spezieller Dank geht an Georg Hofferek und Karin Greimel für ihr Mentoring speziell am Anfang meines Studiums, an Martina Seidl, Florian Lonsing und Uwe Egly dafür, dass sie mich in die aufregende Welt von QBF und SAT eingeführt haben, dem FoREnSiC Team für die sehr produktive Zusammenarbeit, Philipp Pani und Patrick Klampfl für ihre Beiträge zur Implementierung von FoREnSiC und Demiurge, sowie allen Kollegen meiner Arbeitsgruppe für die zahllosen Diskussionen und fruchtbaren Kollaborationen.

Weiters danke ich meiner Elisabeth für ihre Unterstützung, ihr Verständnis und die Energie die sie mir gibt. Nicht zuletzt danke ich auch meinen Eltern Rosa und Heinz und meiner ganzen Familie für ihren Rückhalt.

Diese Arbeit wurde von der Europäische Kommission durch die Projekte DIAMOND (FP7-2009-IST-4-248613) und STANCE (317753) sowie vom Österreichischen Wissenschaftsfonds (FWF) durch das Forschungsnetzwerk RiSE (S11406-N23) finanziell unterstützt.

Robert Könighofer
Graz, Österreich, im September 2015

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
BV	Bitvector Arithmetic
CDCL	Conflict-Driven Clause Learning
CEGIS	Counterexample-Guided Inductive Synthesis
CNF	Conjunctive Normal Form
DNF	Disjunctive Normal Form
DQBF	Dependency Quantified Boolean Formulas
EPR	Effectively Propositional Logic
FOL	First-Order Logic
GUI	Graphical User Interface
GR(1)	Generalized Reactivity of Rank 1
IDE	Integrated Development Environment
LIA	Linear Integer Arithmetic
LTL	Linear Temporal Logic
MAX-SMT	Maximum Satisfiability Modulo Theories
MBD	Model-Based Diagnosis
PCNF	Prenex Conjunctive Normal Form
PDR	Property Directed Reachability
QBF	Quantified Boolean Formula
RHS	Right-Hand Side (of an assignment)
S1S	Monadic Second Order Logic of One Successor
SAT	Satisfiability
SFD	Single-Fault Diagnosis
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
TCAS	Traffic Collision Avoidance System

1 Introduction

1.1 Context and Motivation

Quality assurance constitutes a significant part of the effort for developing new hardware or software products. For hardware, estimates for the verification effort are in the order of 30 to 50 percent [89] of the total development costs. For software, the estimates range between 50 and 75 percent of the total costs [212, page 21], [164, page ix]. Testing [164], where the system under consideration is executed with different inputs in order to detect misbehaviors, is certainly the most wide-spread means of verification. However, for most real-world systems, it is practically infeasible to cover the entire input space with test cases. Hence, for critical applications, testing alone is often not enough to achieve a satisfying level of confidence in the correctness of a system.

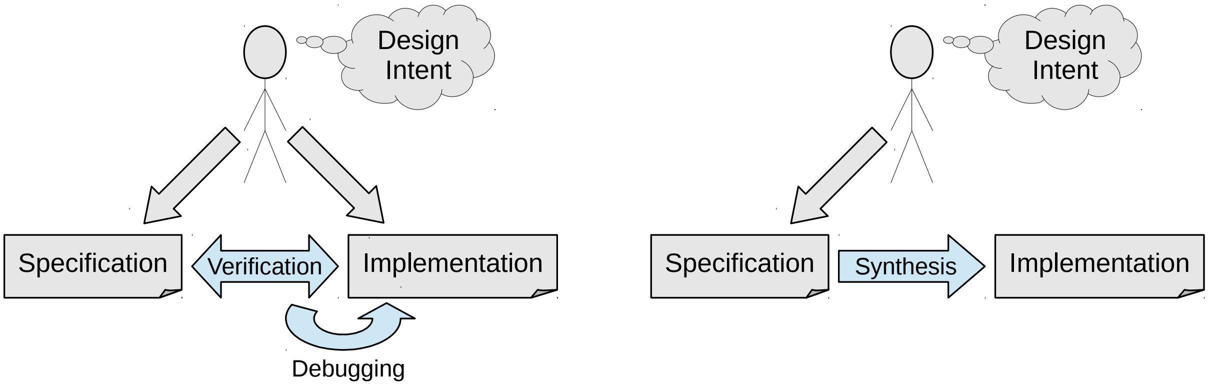
In this section, we discusses how model checking and synthesis techniques can contribute towards a higher system quality with low manual effort. With this motivation in mind, we will then discuss the problems addressed by this thesis, namely the scalability and applicability of synthesis techniques. Finally, we will summarize the contributions of this thesis, which are centered around novel SAT-based synthesis algorithms as well as the application of synthesis in the context of automatic program repair.

1.1.1 Model Checking

Model checking [59, 178, 60, 107] eliminates the incompleteness that is inherent in testing by exhaustively considering all possible input scenarios for a system. The idea is to formally and automatically prove that a given model of a system satisfies a given specification. It is not surprising that this kind of exhaustive analysis comes at a price: the number of inputs and states of the system under analysis can be tremendous, resulting in serious scalability problems. Various techniques have been proposed to overcome this issue. Symbolic model checking [49] avoids enumerating states explicitly and uses formulas as a compact representation of state sets instead. These formulas can in turn be represented using (Reduced Ordered) Binary Decision Diagrams (BDDs) [45], a graph-based representation for propositional formulas. Logical operations are performed by manipulating these graphs. This approach enabled the verification of industrial designs with more than 10^{20} states [49], which was far beyond reach with explicit state representations. However, for certain structures such as barrel shifters or multipliers, BDDs are known to explode in size and thus scale insufficiently [45]. Satisfiability-based Bounded Model Checking (BMC) [24] is an alternative to overcome this limitation. The behavior of the program is encoded into a propositional formula in Conjunctive Normal Form (CNF). A SAT solver is then used to search for an input sequence that results in a specification violation. A bound on the maximum length of the execution is iteratively increased during this search. Craig Interpolation [67] can be used as abstraction and to get unbounded correctness proofs [159]. Another recently proposed model checking algorithm is called[1] IC3 [41]. Like BMC, IC3 also uses a SAT solver to search for incorrect program executions. However, the behavioral description of the system is not unfolded during this search. Instead, over-approximations of reachable states are computed using incremental induction. Currently, IC3 is one of the most powerful model checking algorithms for industrial hardware designs.

Model checking techniques have also been applied successfully to software programs. The tool CBMC [61] implements Bounded Model Checking for C programs in a scalable way. Tools such as SLAM [12], BLAST [20] or SATABS [62] apply abstraction with counterexample-guided refinements in order to reduce the state space. Another interesting technique for software is symbolic execution [63, 136, 51], where the program is executed with symbols that act as placeholders for arbitrary input values. Since this symbolic execution is not necessarily carried out exhaustively, it can be seen as a middle ground

[1] Some authors call the underlying algorithm *Property Directed Reachability (PDR)* [79] and its implementation IC3. However, for simplicity, we will use the term IC3 for both the algorithm and its implementation in this thesis.



(a) Verification-based design flow. **(b)** Synthesis-based design flow.

Figure 1.1: Reduction of the development effort due to synthesis. The left figure shows the typical design flow based on verification: The designer creates an implementation and a formal specification. Formal verification techniques such as model checking are then used to check conformance. Mismatches need to be debugged manually. The right figure shows a flow using synthesis. The designer only needs to create a formal specification. A synthesis algorithm then computes a correct implementation automatically.

between model checking and testing. Concolic execution [99, 191] is a variant of symbolic execution, where the program is executed using concrete inputs, but a symbolic execution for the activated execution path is performed in parallel. The resulting path condition can be used to compute new inputs that trigger a different execution path. This approach scales up to very large programs. In Microsoft's SAGE [100] project, concolic execution is used to find potential security problems in parsers of various applications such as image processors, media players, and operating system components. Since 2008, it has been running all around the clock on more than 100 machines in parallel, and saved millions of dollars by detecting bugs that would have been shipped otherwise [100].

1.1.2 Synthesis

A common criticism of formal verification techniques such as model checking is that they are only applied a posteriori, i.e., after the implementation is already completed. Synthesis is more ambitious: it constructs an implementation from a given declarative specification fully automatically. The specification may only express *what* the system shall do, but not *how* this is to be achieved. Hence, writing a specification can be significantly easier than implementing it. Another advantage is that synthesized implementations are *correct-by-construction*, i.e., guaranteed to satisfy the specification from which they have been constructed. Assuming that the specification expresses the design intent correctly and completely, this eliminates the need for verification and debugging of the implementation, thereby saving development time and costs. This reduction in the development effort is illustrated in Figure 1.1.

Synthesis is a game. Model checking can be understood as (exhaustive) search for inputs under which a (model of the) system violates its specification. That is, the inputs are the only source of non-determinism. Synthesis, on the other hand, needs to handle two sources of non-determinism: the unknown inputs and the (yet) unknown system implementation. Synthesis can thus be seen as a game between two players: The environment player controls the inputs of the system to be synthesized. The system player controls the outputs and attempts to satisfy the specification for *every* environment behavior. The environment player thus has the role of the antagonist, with the objective to violate the specification. This is illustrated in Figure 1.2. The game-based approach to synthesis computes a *strategy* for the system player to win the game (i.e., to satisfy the specification) against every environment player. An implementation of such a winning strategy forms the final solution. Computing a winning strategy naturally involves dealing with alternating quantifiers because for every input (or environment

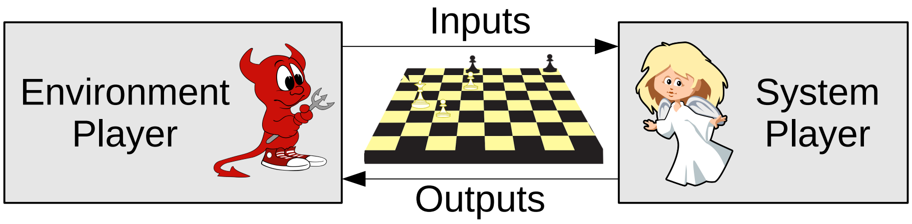


Figure 1.2: Synthesis is a game between two players. The environment player controls the inputs of the system and attempts to violate the specification. The system player controls the outputs and needs to satisfy the specification against every environment behavior.

behavior) there must exist some output (or system behavior) satisfying the specification. This stands in contrast to model checking, where existential quantification suffices.

History. The synthesis vision goes back to 1962, where Alonzo Church [57] first posed the problem for specifications given in so-called *Monadic Second Order Logic of One Successor (S1S)*. S1S is a subset of second order logic, which allows quantification over individual elements and sets (but not over arbitrary predicates), and has one successor relation. A solution to Church's problem was given by Büchi and Landweber [48] a few years later. Unfortunately, the worst-case complexity of the synthesis problem from S1S specifications is non-elementary, i.e., cannot be expressed with a fixed number of exponentiations. Two decades later, Pnueli and Rosner [176] studied the synthesis of so-called *reactive systems* from Linear Temporal Logic (LTL) [175] specifications. Reactive systems interact with their environment via inputs and outputs in a synchronous way: in every time step, the environment provides input values and the system responds with output values. This interaction is carried out ad infinitum, i.e., reactive systems conceptually never terminate. Thus, reactive systems directly model (synchronous) hardware designs. Unfortunately, the problem of synthesizing reactive systems from LTL specifications has a doubly exponential worst-case complexity [186]. Due to these high complexities, synthesis was mainly of academic interest for quite some time.

Recent progress. Despite these high worst-case complexities, recent developments made synthesis techniques applicable to real-world problems. One approach is to set a bound on the size of the systems to construct [87], and increase this bound iteratively, similar to BMC. The rationale behind this approach is that real-world specifications typically have relatively simple implementations. Moreover, the user typically prefers simple implementations, and may not be interested in an overly complicated solution even if one exists. Another direction is to limit the expressiveness of the specification language. For instance, Generalized Reactivity of Rank 1 (GR(1)) [36] is a specification language that is expressive enough for many applications. At the same time, relatively efficient algorithms (with a singly exponential worst-case complexity) and tools [29] implementing them exist. Finally, just like in formal verification, symbolic algorithms are important for achieving scalability in practice. In synthesis of reactive systems, BDDs are the predominant symbolic reasoning engine. This is witnessed for instance by the fact that all submissions to the reactive synthesis competition SyntComp[2] 2014 [117], except for our own, are BDD-based. One reason is that synthesis algorithms inherently deal with alternating quantifiers (see above). BDDs provide both universal and existential quantifier elimination. Moreover, BDDs not only compute one satisfying assignment for a formula, but always represent all satisfying assignments simultaneously. This is exploited by many synthesis algorithms. This dominance of BDDs in synthesis stands in contrast to formal verification, where BDDs have largely been displaced by SAT solvers.

1.1.3 Controller Synthesis

Besides scalability, another challenge that needs to be addressed in synthesis is applicability. Writing a declarative specification for a complete system can be very cumbersome or even unmanageable. The reason is that certain aspects of a system are often easier to define imperatively, i.e., to implement by

[2] `http://www.syntcomp.org/` (last visit on 2015-08-01).

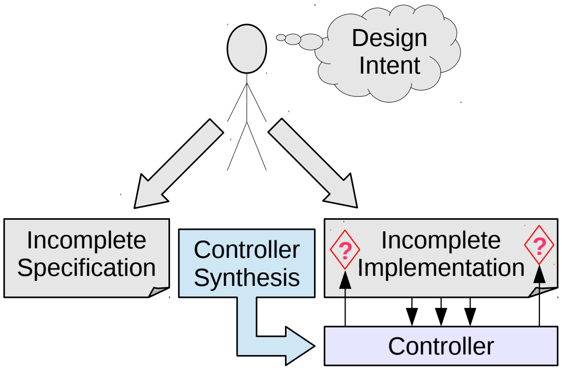
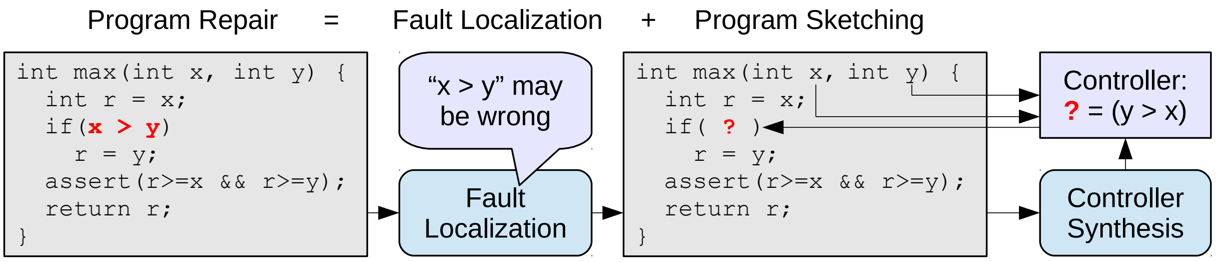


Figure 1.3: Mixed imperative/declarative programming using controller synthesis. The designer implements (most) parts of the system but can leave certain internal variables open (symbolized by question marks). Controller synthesis then completes the implementation with a controller that defines the open variables such that a given specification is satisfied. In contrast to Figure 1.1b, the specification does not have to be complete but can be focused towards the missing parts.

traditional means. Controller synthesis attempts to combine the best of the imperative and the declarative programming paradigms: (most) parts of the implementation are already given by the user. Only the implementation for certain signals is missing and needs to be synthesized such that the entire system satisfies a given specification. This is illustrated in Figure 1.3 and has many interesting applications.

Program sketching. Program sketching [197] allows the user to write programs with “holes” in it. Holes represent yet unknown program parts that are difficult to engineer. They may be as simple as an unknown integer constant, or more complex constructs such as an unknown branch condition. The user also provides a specification, e.g., in form of a reference implementation or assertions in the code. A synthesizing compiler then computes implementations for the holes such that the entire program satisfies the given specification. Obviously, such a mixed imperative/declarative programming paradigm can save development time because difficult aspects can be delegated to the synthesis tool. The rightmost part of Figure 1.4 illustrates this application for a simple software program (in C syntax) that is supposed to compute the maximum of two integer numbers. The programmer left the `if`-condition as a hole to be synthesized. An assertion in the code serves as (incomplete) specification. Synthesis algorithms can be used to compute a controller, which is is a piece of code defining the unknown value of the condition such that the specification is satisfied for all inputs. The example in Figure 1.4 illustrates program sketching for a software program, but the same principle can be applied to hardware as well.

Program repair. Another interesting application of controller synthesis is the automatic repair of incorrect software or hardware programs [128, 140]. Manual debugging is a time-consuming and often frustrating activity. Estimates for hardware [89] say that debugging constitutes around 60 % of the



```
int max(int x, int y) {
  int r = x;
  if(x > y)
    r = y;
  assert(r>=x && r>=y);
  return r;
}
```

```
int max(int x, int y) {
  int r = x;
  if( ? )
    r = y;
  assert(r>=x && r>=y);
  return r;
}
```

Figure 1.4: Program repair and program sketching as applications for controller synthesis. Given an incorrect program and a specification (the assertion), fault localization algorithms can identify program parts that may be erroneous. Replacing these erroneous parts with holes gives a program sketching problem. Controller synthesis methods can synthesize code that controls the values of the holes such that the specification is satisfied.

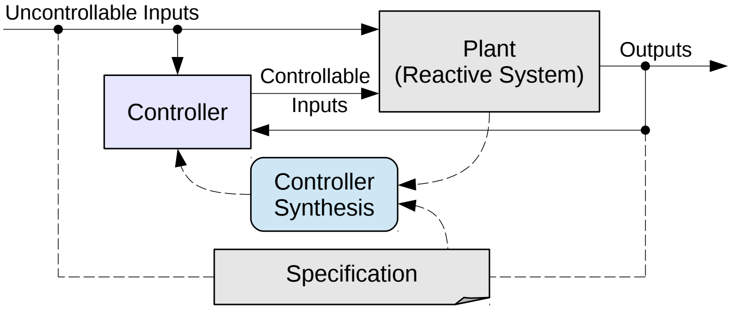


Figure 1.5: Synthesizing a controller for a plant that can be modeled as a reactive system. The plant has uncontrollable inputs that are provided by the environment. The controllable inputs should be defined in such a way that a given specification is satisfied.

verification effort, which is around 30 % of the total development effort. For software, estimates for the debugging effort range from 50 to 80 % of the development and maintenance effort [64]. While there is a broad range of existing techniques and tools to assist the developer in detecting and locating faults, the process of fixing bugs is still mostly done manually [101]. At the same time, fixing a fault is often the most difficult step. There is the danger of eliminating only (some but not all) symptoms, or even introducing new faults. Controller synthesis techniques can be used to suggest potential fixes to the developer, thus assisting in the last an crucial debugging step. A realization of automatic program repair via controller synthesis is illustrated in Figure 1.4. Given a program that violates its specification, fault localization techniques can be used to identify program parts that may be responsible for the incorrectness. These potentially incorrect program parts can then be replaced by holes as used in program sketching. A synthesized controller can finally define the value of the holes in such a way that the specification holds for all inputs, thereby repairing the incorrect program. Figure 1.4 illustrates this flow for software, but the same principle can be applied to hardware programs as well.

Control engineering. Other, more obvious applications of controller synthesis include control engineering, where a given plant needs to be operated in such a way that some specification is satisfied. This setting is illustrated in Figure 1.5. The plant can be given as a reactive system with uncontrollable inputs that are provided by the environment, and controllable inputs that need to be defined by a controller. Synthesis algorithms can be used to construct such a controller automatically from a model of the plant and the specification that must be satisfied. The resulting controller observes the (sequence of) inputs and outputs of the plant and defines the controllable inputs of the plant based on this information.

1.2 Problem Statement

This thesis addresses two main challenges in synthesis: scalability and applicability.

Scalability. Synthesis inherently involves dealing with quantifier alternations. Synthesis algorithms for reactive (hardware) systems are thus traditionally realized using BDDs as the underlying symbolic reasoning engine. Yet, decision procedures for the satisfiability of quantified and unquantified formulas have experienced enormous scalability improvements over the last years and decades. Powerful implementations for different logics are available in the form of SAT solvers, QBF solvers, SMT solvers, etc. These engines cannot only decide the satisfiability of a formula, but also compute satisfying assignments, unsatisfiable cores, or assignments satisfying a maximum number of given constraints. Furthermore, they can be used incrementally to solve sequences of similar queries more efficiently. The questions of (a) how to exploit these features in synthesis effectively and (b) to which extent decision procedures can compete with BDDs in hardware synthesis have not been studied thoroughly before.

Applicability. Synthesis of complete systems from declarative specifications is often unrealistic because (a) writing a complete specification for a large system can be practically infeasible, and (b) synthesis algorithms may not be scalable enough to realize them. A more promising direction is the combination of the imperative and the declarative programming paradigm, where the implementation is largely given but certain aspects of it need to be synthesized such that some specification is satisfied. Solutions to such kinds of problems do not only require efficient synthesis algorithms but also scalable program analysis techniques for the existing parts of the implementation. Furthermore, to keep the human engineer in the loop, synthesis results must be simple and understandable. Solutions for this combination of problems have not been studied extensively, especially in the context of automatic repair of software programs.

Orthogonal challenges. Synthesis challenges that are (mostly) orthogonal to this thesis include (1) improving the quality of synthesized systems, (2) providing techniques for specification engineering and debugging, and (3) providing appropriate specification languages.

1.3 Thesis Statement

By careful utilization of modern features such as incremental solving and the computation of unsatisfiable cores, decision procedures for the satisfiability of formulas can be used to build scalable synthesis algorithms. In combination with flexible program analysis techniques and fine-grained fault localization approaches, such satisfiability-based synthesis algorithms can increase the applicability of synthesis in promising fields like automatic repair of simple software programs.

1.4 Contributions and Publications

This thesis proposes controller synthesis methods based on automatic decision procedures for the satisfiability of quantified and unquantified formulas, both in the hardware- and in the software setting. In the following, we will use the terms "satisfiability-based" or "SAT-based" to indicate the use of such decision procedures. Note that this does not only include SAT solvers [25] for propositional formulas, but also decision procedures for Quantified Boolean Formulas (so-called QBF solvers [97]), Satisfiability Modulo Theories (so-called SMT solvers [14]), and decision procedures for fragments of first order logic. We will write "SAT solver based" to specifically indicate the use of propositional SAT solvers. With this terminology in mind, the main contributions of this thesis can be summarized as follows.

1.4.1 Scalability in Synthesis

To address the scalability challenge, we developed novel algorithms to synthesize reactive hardware controllers from safety specifications using SAT solvers, QBF solvers, and solvers for Effectively Propositional Logic (EPR) [153], which is a decidable subset of first-order logic. The algorithms are designed to effectively exploit solver features such as incremental solving and the computation of unsatisfiable cores. Our synthesis approach consists of two steps. The first step is to compute a strategy to satisfy the specification for all environment behaviors, or to report that no such strategy exists. The second step is to compute a circuit implementing the strategy.

- For strategy computation, we present novel algorithms based on query learning [7], templates fixing the solution structure, and a reduction to EPR [153].
  - We propose a broad range of optimizations to the presented methods. This includes heuristics to partially expand quantifiers, concepts to exploit information about (un)reachable states, as well as low-level optimizations in the formula encoding.
  - We designed a parallelization that combines different methods and different method configurations using multiple threads. The methods do not only run in isolation but share fine-grained information that can support the progress in other threads.
- For computing circuits from strategies, we present solutions based on QBF certification [168], query learning [7], Craig interpolation [67], and a combination of the latter two.

  - Our circuit synthesis methods not only work for safety specifications but also for strategies computed from other specification formalisms. We therefore present the general solutions as well as efficient realizations for the special case of safety synthesis problems.
  - Again, we present several optimizations, including heuristics to exploit variable independencies and optimizations in the formula encoding.
  - A parallelization combining different approaches in different threads is proposed as well.

- We have implemented our methods in an open-source synthesis called Demiurge[3]. Its input is a safety specifications in AIGER[4] format, as specified by the rules for the reactive synthesis competition SyntComp [117]. Demiurge already won two gold medals (one in 2014 and one in 2015) in this synthesis competition. Our tool is highly configurable, e.g., regarding the solvers to use or the optimizations to enable. Demiurge is also designed to be easily extendable, and can thus also be seen as a framework for implementing new methods and optimizations in new back ends.
- We present an extensive experimental evaluation where we compare our different methods and optimizations on the benchmarks from the SyntComp 2014 competition, which includes several specifications for real-world synthesis problems.
  - A comparison with a BDD-based implementation demonstrates that our satisfiability-based approach is competitive in strategy computation. Our parallelization is faster by around one order of magnitude in our experiments. For computing circuits from strategies, our satisfiability-based methods are also faster by more than one order of magnitude and produce circuits that are smaller by even two orders of magnitude on the average in our experiments.
  - We also perform a comparison with AbsSynthe [43], a BDD-based synthesis tool that implements advanced concepts such as abstraction/refinement and won the synthesis track in SyntComp 2014. The performance of our parallelization does not lack far behind AbsSynthe in strategy computation. Our parallelized circuit synthesis can solve more instances and produces circuits that are smaller by one order of magnitude on average in our experiments.
  - On top of these excellent results on the average over our benchmark set, our approach is particularly superior for certain benchmark classes. We thus conclude that our satisfiability-based methods form a valuable complement to existing synthesis approaches.

1.4.2 Applicability of Synthesis

To address the applicability challenge, we developed a novel approach for the automatic repair of incorrect software programs specified by assertions in the code. Our approach consists of three steps: program analysis to lift the repair problem into the domain of logic, fault localization to identify candidate program parts for repair, and finally the repair synthesis itself. SMT solving is used as the main underlying reasoning technology.

- For program analysis, we propose a solution based on symbolic [63, 136] or concolic [99, 191] execution. It allows for incomplete analysis with various parameters to trade accuracy for efficiency.
- Based on the diagnostic information collected by the program analysis step, we present a fine-grained fault localization approach using the fault model of incorrect expressions and ideas from model-based diagnosis [182]. An SMT solver is finally used to compute potential fault locations.
- As an alternative to this SMT-based fault localization approach, we also work out a solution based on deductive verification and first-order theorem proving. Instead of assertions, it uses pre- and postconditions as a specification.
- We present a template-based method to synthesize human-readable replacements of the potentially incorrect expressions identified by automatic fault localization. It is based on Counterexample-Guided Inductive Synthesis (CEGIS) [197, 196], which iteratively refines repair candidates based on counterexamples, and uses an SMT solver as the underlying reasoning engine.

[3] `www.iaik.tugraz.at/content/research/opensource/demiurge` (last visit on 2015-08-01).

[4] `http://fmv.jku.at/aiger/` (last visit on 2015-08-01).

  - We also propose heuristics to speed up the CEGIS algorithm for repair synthesis using Maximum Satisfiability Modulo Theories (MAX-SMT) [162] solving.
  - We introduce an improved repair synthesis method that performs program analysis on the fly and only for specific counterexamples that are encountered in the CEGIS process. It also decouples the repair candidate computation from the candidate verification and thereby allows for a broader range of verification techniques and specification formats, including test case execution and model checking.

- We have implemented our repair synthesis approach as a proof of concept for simple C programs in the tool FoREnSiC[5], which is available under an open-source license. Our implementation is highly configurable, e.g., regarding the logic and the engine to use for SMT solving, the accuracy of program analysis and fault localization, as well as various heuristics and optimizations. Our fault localization variant using deductive verification has been prototyped as an extension to the widely used software analysis tool suite Frama-C [69].
- In an experimental evaluation, we demonstrate that our software repair approach can produce helpful diagnostic information also for non-trivial programs in reasonable time. A comparison of our fault localization approach with the existing tool Bug-Assist [130] indicates a better diagnostic resolution. Moreover, our repair synthesis approach turned out to scale much better than the existing program sketching tool Sketch [196, 197] in our experiments. We thus conclude that automatic program repair is a promising application for satisfiability-based controller synthesis techniques.

1.4.3 List of Publications

The rules of the Doctoral School of Computer Science at Graz University of Technology require that every dissertation contains a list of publications of the candidate, which explains (a) how the individual publications are related to the dissertation, and (b) the origin of contributions if papers have been written collaboratively. This information is provided in the following. Additionally, every section of this thesis will list the publications it is based on.

In summary, this thesis is directly based on 7 conference publications. All in all, I contributed to 15 conference publications, 2 workshop publications and 2 journals by August 2015. Many of them are closely related to this thesis. Furthermore, I presented 13 of these publications at conferences and workshops. The following list of publications is sorted thematically rather than chronologically.

This thesis is directly based on the following 7 publications:

VMCAI'14 [39]: Hardware synthesis algorithms usually work in two steps. The first step is to compute a strategy to enforce the specification. The second step is to implement this strategy in a circuit. This paper presents satisfiability-based methods to solve the first of these two steps for the case of safety specifications. Hence, this paper is concerned with the scalability challenge addressed by this thesis. The paper was mainly written by myself. The development of the synthesis algorithms, the implementation, and the experiments were mostly conducted by myself as well. Roderick Bloem contributed ideas for algorithms and optimizations in countless discussions and proofread the paper. Martina Seidl contributed to a more efficient encoding into QBF, and by implementing an extension of the QBF preprocessor Bloqqer [189], which was used for some methods in the experiments. She also proofread the paper. Aaron R. Bradley contributed to the reachability optimization through discussions about using concepts from the model checking algorithm IC3 [41] in synthesis. Discussions with Andreas Morgenstern helped in the reimplementation of an alternative SAT-based method [163], which served as baseline for the comparison in the paper. The students Fabian Tschiatschek and Mario Werner contributed a BDD-based implementation which also served as baseline for the comparison. Finally, Bettina Könighofer helped creating benchmarks. I presented the paper at the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) in San Diego, USA, on the 19th of January 2014.

[5] http://www.informatik.uni-bremen.de/agra/eng/forensic.php (last visit on 2015-08-01).

FMCAD'12 [82]: This paper presents learning-based methods to solve the second step of a typical hardware synthesis procedure, namely the computation of circuits implementing a strategy. The algorithms are independent of the underlying symbolic reasoning engines, but have been implemented using BDDs for the experiments in the paper. Thus, this work does not directly deal with satisfiability-based synthesis methods. However, the presented algorithms have later been used with SAT- and QBF solvers (in VMCAI'14 [39] and FMCAD'14a [31]). The algorithms and a first version of the implementation were mostly developed by Rüdiger Ehlers. Georg Hofferek contributed a discussion of alternative approaches and practical experience with them. I focused on optimizations and experiments. The work of writing the paper was split quite evenly.

FMCAD'14a [31]: This paper presents satisfiability-based methods to solve the same problem as the previous paper, namely the computation of circuits implementing a strategy. This paper therefore directly addresses the scalability challenge outlined in the problem statement. The algorithms, optimizations, implementation, experiments, and the paper writing work was mainly done by myself. However, the interpolation-based method was implemented and evaluated by Patrick Klampfl in the course of his Bachelor's Thesis under my supervision. Patrick Klampfl also developed the dependency optimization presented in the paper. Uwe Egly and Florian Lonsing contributed towards efficient utilization of incremental QBF solving in one of the presented methods, and both proofread the paper. Roderick Bloem contributed ideas and also proofread the paper. I presented the paper at Formal Methods in Computer-Aided Design (FMCAD) in Lausanne, Switzerland, on the 22nd of October 2014.

FMCAD'11 [140]: This paper presents an approach for automatic fault localization and correction in simple software programs, and mainly addresses the applicability challenge from the problem statement. The concept was developed by myself with guidance and supervision by Roderick Bloem. The implementation, optimizations, experiments, and paper writing work was performed by myself. Roderick Bloem proofread the paper. I presented the paper at Formal Methods in Computer-Aided Design (FMCAD) in Austin, USA, on the 31st of October 2011.

HVC'12a [141]: This paper presents a program repair approach that performs program analysis on the fly. It can be seen as an improvement of the previous publication [140]. Again, the algorithms, implementation, experiments, and paper writing work was mainly done by myself. Roderick Bloem supervised this work and proofread the paper. I presented the paper at the 8th Haifa Verification Conference (HVC) in Haifa, Israel, on the 6th of November 2012.

HVC'12b [30]: This paper presents the tool FoREnSiC, which implements the fault localization and correction methods presented in the previous two papers [140, 141]. FoREnSiC consists of a front end, an internal model of the program, and three back ends. I was involved in the design and implementation of the internal model. Furthermore, I designed and implemented one of the three back ends, and wrote the corresponding sections of the paper. I also presented the paper at the 8th Haifa Verification Conference (HVC) in Haifa, Israel, on the 6th of November 2012.

HVC'14 [145]: This paper discusses an alternative fault localization approach for software programs. While the FMCAD'11 [140] publication uses symbolic or concolic execution and model-based diagnosis, this paper follows an approach based on deductive verification and theorem proving. The concept, the implementation, and the experiments were mostly performed by myself. Loïc Correnson gave support for our proof-of-concept implementation in Frama-C. Ronald Tögl helped writing the paper. Roderick Bloem supervised the work and proofread the paper. I presented the work at the 10th Haifa Verification Conference (HVC) in Haifa, Israel, on the 19th of November 2014.

I also presented our work on QBF-based synthesis algorithms [39, 31, 189] in an invited talk at the QBF Workshop in Vienna, Austria, on the 13th of July 2014. Furthermore, together with Alexander Finder, I presented the FoREnSiC tool [30] at a University Booth at Design, Automation & Test in Europe (DATE) in Dresden, Germany, from the 13th to the 15th of March 2012.

In addition to the content of these peer-reviewed publications, this thesis also contains a new heuristic for partial quantifier elimination in SAT-based hardware controller synthesis, additional correctness proofs, and more extensive experimental results.

The following publications are more loosely related to this thesis:

DATE'14 [189]: This paper presents an extension of the QBF preprocessor Bloqqer to preserve satisfying assignments, which is a prerequisite for using QBF preprocessing in synthesis. Thus, this paper also addresses the scalability challenge mentioned in the problem statement. Results of this work have been used in the VMCAI'14 [39] and FMCAD'14a [31] publications mentioned earlier. Martina Seidl did the main work regarding theory, implementation, and paper writing. I mainly contributed experimental results from the domain of synthesis and wrote the corresponding sections of the paper.

TACAS'15a [28]: This paper presents a program sketching approach for concurrent reactive programs based on the existing notion of Assume-Guarantee Synthesis [55], but extended with partial information constraints. Since the problem is undecidable in general, the paper presents a semi-decision procedure based on bounded synthesis [87] and SMT solving. Hence, this work also deals with satisfiability-based methods for controller synthesis, in the application of program sketching. However, it is not included in the core publications of this thesis, because it is not primarily my own work. The concept has been developed by Swen Jacobs. The complexity results were mainly worked out by Krishnendu Chatterjee. I implemented the method, performed the experiments, and wrote the corresponding sections of the paper. I also presented the work at the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) in London, UK, on the 16th of April 2015.

TACAS'15b [37]: This paper presents an approach to synthesize a "safety shield" that enforces critical properties of a reactive system at runtime if the properties cannot be verified statically. Similar to our work on program repair and sketching, it combines existing implementations with synthesized code, i.e., supports the mixed imperative/declarative engineering paradigm. The problem and initial ideas for a solution were contributed by Chao Wang. The final solution for safety specifications was mainly developed by myself and Roderick Bloem. Bettina Könighofer implemented the approach and performed experiments. I presented the paper at the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) in London, UK, on the 16th of April 2015.

FMCAD'14b [35]: This paper presents Bettina Könighofer's work on synthesis of atomic sections in concurrent software programs such that no race conditions can occur. Thus, this work also deals with a mixed imperative/declarative programming paradigm, but in the context of concurrent programs. My contribution to this paper was mostly in co-supervising the students Simon Ausserlechner and Raphael Spörk, and in improving the presentation in the paper. I also presented the paper at Formal Methods in Computer-Aided Design (FMCAD) in Lausanne, Switzerland, on the 22nd of October 2014.

FMCAD'09 [142]: This paper summarizes my Master's Thesis [139]. It addresses the problem of debugging unrealizable or unintended specifications using synthesized counterstrategies and is thus orthogonal to the topics addressed by this thesis. The paper has been written with extensive support from Georg Hofferek and Roderick Bloem. I presented the paper at Formal Methods in Computer-Aided Design (FMCAD) in Austin, USA, on the 18th of November 2009.

HVC'10 [143]: This paper presents an extension to the solution for specification debugging given in the FMCAD'09 [142] publication: techniques from model-based diagnosis are used to explain the reasons for unrealizability of a specification to the user. The approach was developed and implemented mainly by myself, with support from Georg Hofferek and Roderick Bloem, especially in writing the paper. I presented the work at the 6th Haifa Verification Conference (HVC) on the 6th of October 2010.

CAV'10 [29]: This paper presents the requirement analysis and synthesis tool Ratsy. This tool implements the specification debugging techniques of FMCAD'09 [142] and HVC'10 [143], and I contributed to the corresponding sections of the paper. I presented Ratsy at the 22nd International Conference on Computer Aided Verification (CAV) in Edinburgh, UK, on the 18th of July 2010.

STTT'13 [144]: This journal article summarizing the previous three publications [142, 143, 29] in a combined flow. The article has mainly been written by myself, with presentation improvements by Georg Hofferek and proofreading by Roderick Bloem.

SYNT'12 [34]: This paper is about Bettina Könighofer's work on synthesis of robust systems. This work deals with the quality of synthesized systems and is thus orthogonal to this thesis. My contribution to this work is mostly in co-supervising Bettina Könighofer and helping to present the work in the paper.

Acta'14 [27]: This journal article summarizes existing work on synthesis of robust systems, including the SYNT'12 [29] publication. My contribution to this article was mainly the illustration with examples and co-developing a combined approach for safety and liveness specifications.

SYNT'14 [32]: This paper raises the concern that existing synthesis approaches do not always handle assumptions in specifications in a desirable way and surveys existing work on this topic. Hence, the paper is mostly about the quality of synthesized systems and is thus orthogonal to the problems addressed by this thesis. The paper has been written by all authors in approximately equal parts. I presented this work at the 3$^{\text{rd}}$ Workshop on Synthesis in Vienna, Austria, on the 23$^{\text{rd}}$ of July 2014.

QSIC'14 [38]: This paper presents Franz Röck's work on automatic completion of given test suites. The connection to this thesis is that the underlying methods, namely symbolic execution, are also used in our program repair approach. My main contribution to this paper was in improving the presentation.

1.5 Structure of this Thesis

This thesis is structured as follows. Chapter 2 conveys background knowledge that is helpful for understanding the contributions of this thesis. This includes various logics and corresponding reasoning engines, a baseline approach for hardware synthesis, algorithms for query learning and counterexample-guided synthesis that will be used in our SAT-based synthesis algorithms, as well as software program analysis techniques and a fault localization concept for our program repair approach. Chapter 2 also serves the purpose of introducing notation that will be used throughout this thesis.

Chapter 3 introduces our satisfiability-based approach for hardware controller synthesis from safety specifications. It consists of two steps: the computation of a strategy, and the computation of a circuit implementing this strategy. Our algorithms and optimizations for these two steps will be presented in Section 3.1 and Section 3.2, respectively. Section 3.3 will then discuss our implementation and experimental evaluation.

Chapter 4 deals with satisfiability-based controller synthesis in the application of automatic program repair. The approach we propose here consists of three steps: program analysis, fault localization, and repair synthesis. Our basic solutions for these steps will be presented in the Sections 4.2, 4.3 and 4.5, respectively. Section 4.4 describes our alternative solution for fault localization using deductive verification. Our improved repair synthesis flow with on-the-fly program analysis is worked out in Section 4.7. Section 4.6 presents a variant that uses test cases as a specification, and Section 4.8 discusses other variations and parameters. Our implementation and evaluation is finally presented in Section 4.9.

In Chapter 5, we compare our contributions to related work. Chapter 6 finally concludes the thesis and gives suggestions for future work.

2 Background and Notation

Parts of this chapter are based on the previous publications of the author on which this thesis is based [82, 39, 31, 140, 141, 145, 30]. References to these sources are not always made explicit.

This section serves two purposes. On the one hand, it introduces background knowledge that is helpful for understanding the remainder of this thesis. On the other hand, this section introduces notation that will be used throughout this thesis. All abbreviations that we introduce are summarized on Page xix. An index of the main terms can be found on Page 182.

After some basic notation, we will introduce various logics and corresponding reasoning engines. Section 2.4 will then explain how large sets can be represented symbolically as formulas in these logics. Based on these symbolic representations, Section 2.5 will discuss the standard concepts for hardware synthesis with a focus on safety specifications. Next, we present query learning and Counterexample-Guided Inductive Synthesis (CEGIS), two concepts that will be used at various occasions in our SAT-based synthesis algorithms. We finally turn to methods for software program analysis as well as fault localization via Model-Based Diagnosis (MBD), which will both be used in our application of controller synthesis in automatic program repair.

2.1 Basic Notation

We denote the Boolean domain by $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, the set of natural numbers (including 0) by $\mathbb{N}$, the set of integer numbers by $\mathbb{Z}$, and the set of character strings by $\mathbb{S}$. In general, we will use upper case letters for sets, lower case letters for set elements, and calligraphic fonts for tuples defining more complex structures. We will write *iff* as a shorthand for “if and only if”.

2.2 Logics

We will use various kinds of logics to solve synthesis problems. This section introduces these logics. Decision procedures and reasoning engines for these logics will then be introduced in Section 2.3.

Variables and formulas. We will use lower case letters for variables and capital letters to denote formulas. Recall that capital letters are also used to denote sets, but this is no coincidence since we will later use formulas to represent sets (see Section 2.4). Vectors of variables will be written with an overline. For clarity, we will often write the variables that occur freely in a formula in brackets. For instance, $F(\overline{x})$ denotes a formula over the variables $\overline{x} = (x_1, x_2, \dots, x_n)$. If the variables are clear from the context, we will sometimes omit the brackets, i.e., write only F instead of $F(\overline{x})$. Furthermore, we will use the brackets to denote variable substitutions: if $F(\dots, x, \dots)$ is a formula, we denote by $F(\dots, y, \dots)$ the same formula but with all occurrences of x replaced by y. With a slight abuse of notation, we will also treat vectors of variables like sets if the order of the elements is irrelevant. For instance, $\overline{x} \cup \overline{y}$ denotes a concatenation of two variable vectors, and $\overline{x} \setminus \{x_i\}$ denotes the variable vector $\overline{x}$ but with element x_i removed.

Operator precedence. Save for cases where too many brackets hamper readability, we will avoid ambiguities in operator precedence. However, for the avoidance of doubt, will will use the following precedence order (from stronger to weaker binding) for operators in formulas: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists$.

2.2.1 Propositional Logic

All variables in propositional logic are Boolean, i.e., take values from the domain $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. We will use the standard Boolean connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, encoding negation, conjunction, disjunction, implication, and equivalence, respectively.

Conjunctive Normal Forms (CNFs). A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. We will sometimes treat clauses and cubes as sets of literals. For instance, given that l is a literal and c_1, c_2 are clauses, we write $l \in c_1$ to denote that l occurs as a disjunct in clause c_1, and we write $c_1 \subseteq c_2$ to denote that all literals of clause c_1 also occur in clause c_2. A propositional formula is in *Conjunctive Normal Form (CNF)* if it is written as a conjunction of clauses. There are two reasons why CNF representations are important. First, decision procedures for satisfiability usually require the input formula to be in CNF. Second, every formula can be transformed into an equisatisfiable formula in CNF representation by introducing at most a linear amount of auxiliary variables. This transformation is called *Tseitin transformation* [208]. An improvement by exploiting the polarity (even or odd number of negations) of subformulas to obtain smaller CNF encodings has been proposed by Plaisted and Greenbaum [174].

Variable assignments. We will use cubes to describe (potentially partial) truth assignments to variables: unnegated variables of the cube are set to true, negated ones are false. We will use bold letters to denote cubes. For instance, $\mathbf{x}$ denotes a cube over the variables $\overline{x}$. An *$\overline{x}$-minterm* is a cube that contains all variables of $\overline{x}$ either negated or unnegated (but not both). Thus, minterms describe *complete* assignments to Boolean variables. We write $\mathbf{x} \models F(\overline{x})$ to denote that the $\overline{x}$-minterm $\mathbf{x}$ satisfies the formula $F(\overline{x})$. Given a formula $F(\ldots, \overline{x}, \ldots)$ and an $\overline{x}$-minterm $\mathbf{x}$, we write $F(\ldots, \mathbf{x}, \ldots)$ to denote the formula F but with all occurrences of the variables $\overline{x}$ replaced by their respective truth value defined by $\mathbf{x}$.

Unsatisfiable cores. Let F be an unsatisfiable formula in CNF. A *clause-level unsatisfiable core* is a subset of the clauses of F that is still unsatisfiable. While this definition is widely used, many applications require the minimization of "interesting" constraints while the remaining constraints remain fixed. For such problems, Nadel [165] coined the term *high-level unsatisfiable core*. To support such high-level unsatisfiable cores, we use the following definition in this thesis. Let $\mathbf{x}$ be a cube and let $F(\overline{x}, \overline{y})$ be a formula such that $\mathbf{x} \wedge F$ is unsatisfiable. An *unsatisfiable core* of $\mathbf{x}$ with respect to F is a subset $\mathbf{x}' \subseteq \mathbf{x}$ of the literals in $\mathbf{x}$ such that $\mathbf{x}' \wedge F$ is still unsatisfiable. An unsatisfiable core $\mathbf{x}'$ is *minimal* if no proper subset $\mathbf{x}''$ of $\mathbf{x}'$ makes $\mathbf{x}'' \wedge F$ unsatisfiable. With this definition, high-level unsatisfiable cores can be computed by adding conjuncts of the form $x_i \rightarrow G(\overline{y})$ for $x_i \in \overline{x}$ to $F(\overline{x}, \overline{y})$. This way, the constraint $G(\overline{y})$ can be enabled or disabled via the truth value of x_i. Moreover, this notion of unsatisfiable cores is directly supported by many solver.

Interpolants. Let $A(\overline{x}, \overline{y})$ and $B(\overline{x}, \overline{z})$ be two propositional formulas such that $A \wedge B$ is unsatisfiable, and $\overline{y}$ and $\overline{z}$ are disjoint. A *Craig interpolant* [67] is a formula $I(\overline{x})$ such that $A \rightarrow I \rightarrow \neg B$. Intuitively, the interpolant is a formula that is weaker than A, but still strong enough to make $I \wedge B$ unsatisfiable. In addition to that, the interpolant references only the variables $\overline{x}$ that occur both in A and in B.

Cofactors. Let $F(\ldots, x, \ldots)$ be a propositional formula. The *positive cofactor* of F regarding x is the formula $F(\ldots, \mathsf{true}, \ldots)$, where all occurrences of x have been replaced by true. Analogously, the *negative cofactor* of F regarding x is the formula $F(\ldots, \mathsf{false}, \ldots)$.

2.2.2 Quantified Boolean Formulas

Quantified Boolean Formulas (QBFs) [138] extend propositional logic with universal (denoted $\forall$) and existential (denoted $\exists$) quantification of variables. The quantifiers have their expected semantics: Since propositional variables can only be either true or false, $\exists x_i : F(\ldots, x_i, \ldots)$ can be seen as a shorthand for $F(\ldots, \mathsf{true}, \ldots) \vee F(\ldots, \mathsf{false}, \ldots)$. Likewise, $\forall x_i : F(\ldots, x_i, \ldots)$ is short for $F(\ldots, \mathsf{true}, \ldots) \wedge F(\ldots, \mathsf{false}, \ldots)$. Using these rules, a QBF can always be transformed into a purely propositional formula. However, this usually causes a significant blow-up in formula size.

PCNFs. A QBF is in *Prenex Conjunctive Normal Form (PCNF)* if it is written in the form

$$Q_1\overline{x}_1 : Q_2\overline{x}_2 : \dots Q_k\overline{x}_k : F(\overline{x}_1, \overline{x}_2, \dots, \overline{x}_k),$$

where $Q_i \in \{\forall, \exists\}$ and F is a propositional formula in CNF. In this formulation, we use $Q_i\overline{x}_i$ as a shorthand for $Q_i x_{i,1} : \dots Q_i x_{i,n}$ with $\overline{x}_i = (x_{i,1}, \dots, x_{i,n})$. We refer to $Q_1\overline{x}_1 : Q_2\overline{x}_2 : \dots Q_k\overline{x}_k$ as the *quantifier prefix* and call $F(\overline{x}_1, \overline{x}_2, \dots, \overline{x}_k)$ the *matrix* of the PCNF. We require every PCNF to be *closed* in the sense that all variables occurring in the matrix must be quantified either existentially or universally. Hence, a QBF in PCNF can only be valid (equivalent to true) or unsatisfiable (equivalent to false).

Skolem functions. Let

$$\exists\overline{a}_1 : \forall\overline{b}_1 : \dots \exists\overline{a}_k : \forall\overline{b}_k : \exists\overline{c} : Q_1\overline{d}_1 : \dots Q_l\overline{d}_l : F(\overline{a}_1, \overline{b}_1, \dots, \overline{a}_k, \overline{b}_k, \overline{c}, \overline{d}_1, \dots, \overline{d}_l)$$

with $Q_i \in \{\forall, \exists\}$ be a QBF in PCNF that is valid. A *Skolem function* for the existentially quantified variables $\overline{c}$ is a function $f : 2^{|\overline{b}_1|} \times \dots \times 2^{|\overline{b}_k|} \rightarrow 2^{|\overline{c}|}$ that defines the values of the variables $\overline{c}$ based on the universally quantified variables $\overline{b}_1, \dots, \overline{b}_k$ occurring before $\overline{c}$ in the quantifier prefix such that

$$\exists\overline{a}_1 : \forall\overline{b}_1 : \dots \exists\overline{a}_k : \forall\overline{b}_k : Q_1\overline{d}_1 : \dots Q_l\overline{d}_l : F\big(\overline{a}_1, \overline{b}_1, \dots, \overline{a}_k, \overline{b}_k, f(\overline{b}_1, \dots, \overline{b}_k), \overline{d}_1, \dots, \overline{d}_l\big)$$

is still valid. The function f can be seen as a *certificate* to show that values for the variables $\overline{c}$ making the QBF true exist (for any values of the variables $\overline{b}_1, \dots, \overline{b}_k$). Note that f cannot depend on the variables $\overline{d}_1, \dots, \overline{d}_l$ occurring after $\overline{c}$ in the quantifier prefix, independent of whether some $\overline{d}_i$ is quantified universally or existentially.

Herbrand functions. A Herbrand function is the dual of a Skolem function for the case of a QBF that is unsatisfiable. Let

$$\exists\overline{a}_1 : \forall\overline{b}_1 : \dots \exists\overline{a}_k : \forall\overline{b}_k : \forall\overline{c} : Q_1\overline{d}_1 : \dots Q_l\overline{d}_l : F(\overline{a}_1, \overline{b}_1, \dots, \overline{a}_k, \overline{b}_k, \overline{c}, \overline{d}_1, \dots, \overline{d}_l)$$

be such a QBF that is unsatisfiable. A *Herbrand function* for the universally quantified variables $\overline{c}$ is a function $f : 2^{|\overline{a}_1|} \times \dots \times 2^{|\overline{a}_k|} \rightarrow 2^{|\overline{c}|}$ that defines the values of the variables $\overline{c}$ based on the existentially quantified variables $\overline{a}_1, \dots, \overline{a}_k$ occurring before $\overline{c}$ in the quantifier prefix in such a way that

$$\exists\overline{a}_1 : \forall\overline{b}_1 : \dots \exists\overline{a}_k : \forall\overline{b}_k : Q_1\overline{d}_1 : \dots Q_l\overline{d}_l : F\big(\overline{a}_1, \overline{b}_1, \dots, \overline{a}_k, \overline{b}_k, f(\overline{b}_1, \dots, \overline{b}_k), \overline{d}_1, \dots, \overline{d}_l\big)$$

is still unsatisfiable.

Universal expansion. Let $G = Q_1\overline{x}_1 : \dots Q_k\overline{x}_k : \forall y : \exists\overline{z} : F(\overline{x}_1, \dots, \overline{x}_k, y, \overline{z})$ be a QBF in PCNF. The *universal expansion* [47] of variable y in G is the formula $G' =$

$$Q_1\overline{x}_1 : \dots Q_k\overline{x}_k : \exists\overline{z}, \overline{z}' : F(\overline{x}_1, \dots, \overline{x}_k, \mathsf{true}, \overline{z}) \wedge F(\overline{x}_1, \dots, \overline{x}_k, \mathsf{false}, \overline{z}'),$$

where $\overline{z}'$ is a fresh copy of the variables $\overline{z}$. This transformation is equivalence preserving [47]. In our formulation, the universally quantified variable y to expand must only be followed by existential quantifications in the prefix. The variables $\overline{z}$ may depend on y in G, i.e., may take different values for different truth values of y. Hence, they need to be renamed in one copy of the matrix when turning the universal quantification into a conjunction. Note that G' is in PCNF again because the conjunction of two CNFs is again a CNF.

One-point rule. Let $\mathbf{x}$ be an $\overline{x}$-minterm. We have that

$$\Big(\forall\overline{x} : \mathbf{x} \rightarrow F(\overline{x}, \overline{y})\Big) \leftrightarrow \Big(F(\mathbf{x}, \overline{y})\Big) \leftrightarrow \Big(\exists\overline{x} : \mathbf{x} \wedge F(\overline{x}, \overline{y})\Big) \qquad (2.1)$$

holds true because, in all three formulations, F has to hold for a given $\overline{y}$-assignment if and only if the variables $\overline{x}$ have the specific truth values defined by $\mathbf{x}$. A slightly more complicated instance of this rule can be formulated as follows. Let $T(\overline{z}, \overline{x})$ be a formula that defines the variables $\overline{x}$ uniquely based on the values of some other variables $\overline{z}$. Formally, we assume that $\forall \overline{z} : \exists \overline{x} : T(\overline{z}, \overline{x})$ and $\forall \overline{z}, \overline{x}_1, \overline{x}_2 : \big(T(\overline{z}, \overline{x}_1) \wedge T(\overline{z}, \overline{x}_2)\big) \rightarrow (\overline{x}_1 = \overline{x}_2)$. We have that

$$\Big(\forall \overline{x} : T(\overline{z}, \overline{x}) \rightarrow F(\overline{x}, \overline{y})\Big) \leftrightarrow \Big(\exists \overline{x} : T(\overline{z}, \overline{x}) \wedge F(\overline{x}, \overline{y})\Big) \tag{2.2}$$

holds true because for a given $\overline{z}$-assignment $\mathbf{z}$ and a given $\overline{y}$-assignment $\mathbf{y}$, F needs to hold only for the $\overline{x}$-assignment $\mathbf{x}$ that is uniquely defined by T in both formulations. We will use the dualities of Equation 2.1 and 2.2 in various proofs and transformations throughout this thesis.

2.2.3 First-Order Logic

First-Order Logic (FOL) [115] is a more expressive logic, which enables reasoning about elements from arbitrary domains. Let $\mathbb{D}$ be a (potentially infinite) domain and let $\overline{x} = (x_1, x_2, \dots, x_k)$ be variables ranging over this domain. Furthermore, let $\overline{y} = (y_1, y_2, \dots, y_l)$ be Boolean variables ranging over $\mathbb{B}$, let $f_1, f_2, \dots, f_m$ be function symbols and let $p_1, p_2, \dots, p_n$ be predicate symbols. Each function symbol and each predicate symbol has a certain *arity*, i.e., number of arguments to which it can be applied. A *term* in first-order logic is either a domain variable x_i (with $1 \leq i \leq k$) or a function application $f_i(t_1, \dots, t_a)$, where f_i is a symbol for a function with arity a, and all t_i (with $1 \leq i \leq a$) are terms. Intuitively, a term evaluates to an element of domain $\mathbb{D}$. An *atom* is either a propositional variable y_i (with $1 \leq i \leq l$) or a predicate application $p_i(t_1, \dots, t_a)$ where p_i is a symbol for a predicate with arity a, and all t_i (with $1 \leq i \leq a$) are terms. Thus, intuitively, an atom evaluates to a truth value from $\mathbb{B}$. Finally, a *First-Order Logic (FOL)* formula is one of

$$a, \quad \neg F_1, \quad F_1 \vee F_2, \quad F_1 \wedge F_2, \quad F_1 \rightarrow F_2, \quad F_1 \leftrightarrow F_2, \quad \exists x_i : F_1, \text{ or } \quad \forall x_i : F_1,$$

where F_1 and F_2 are First-Order Logic formulas themselves and a is an atom. The semantics of the Boolean connectives and the quantifiers are as expected. A *model* of a FOL formula is a structure that satisfies the formula. It consists of concrete values for all variables that are not explicitly quantified, as well as concrete realizations of all functions f_i and predicates p_i. Similar to propositional logic, we refer to an atom or the negation of an atom as a *first-order literal* . A *first-order clause* is a disjunction of first-order literals. A *first-order CNF* is a conjunction of first-order clauses. A FOL formula is *quantifier-free* if it contains no occurrences of $\exists$ and $\forall$.

In this thesis, we will focus on two subsets of FOL: quantifier-free FOL formulas equipped with theories, and Effectively Propositional Logic (EPR). We will elaborate on these two subsets in the following.

2.2.4 Theories in First-Order Logic

Intuitively, a theory in first-order logic defines a set of predefined function symbols and predicate symbols and equips them with some semantics, which is usually defined via axioms over the predefined symbols. A model for a FOL formula modulo (means here "with respect to") a given theory is a model for the formula *and* all theory axioms.

Example 1. A theory of family relationships may define the binary predicates parentOf and childOf together with the axiom $\forall x, y : \mathsf{parentOf}(x, y) \leftrightarrow \mathsf{childOf}(y, x)$. The FOL formula $\mathsf{childOf}(\mathrm{Adam}, \mathrm{Eve}) \wedge \forall x : \neg\mathsf{parentOf}(\mathrm{Eve}, x)$ would be satisfiable by itself, but is unsatisfiable modulo our theory of family relationships. The formula says that Adam is a child of Eve, and Eve is the parent of nobody, with Adam and Eve being functions of arity 0, i.e., constants.

In this thesis, we will mainly use two theories: the theory of linear integer arithmetic and the theory of (fixed-size) bitvector arithmetic.

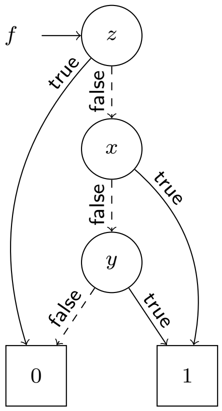


Figure 2.1: A BDD representing the function $f = (x \lor y) \land \neg z$. The root node is marked by an incoming arrow. Terminal nodes are drawn as boxes, non-terminal nodes as circles.

Linear Integer Arithmetic (LIA). The theory of linear integer arithmetic [14] (commonly abbreviated as LIA) defines constants (function symbols with arity 0) for all integers $z \in \mathbb{Z}$.[1] Furthermore, it defines the binary functions $+$ and $-$ with their expected semantics. Note that the $+$ operator can also be used to express multiplications by a constant factor. Finally, the predicates $\leq$ and $=$ are available with their expected semantics. In combination with negation, these predicates can also be used to define $<$, $>$ and $\geq$. This theory is also referred to as *Presburger arithmetic* although, strictly speaking, Presburger arithmetic is defined over $\mathbb{N}$ and without subtraction. However, transformations between $\mathbb{Z}$ and $\mathbb{N}$ are straightforward [14].

Bitvector Arithmetic (BV). The theory of (fixed-size) bitvector arithmetic [14] (abbreviated as BV) defines constants for all bitvectors of a given size n. Furthermore, following the definition of the SMT-LIB standard [13], all operators that are usually available in imperative programming languages are defined with their expected semantics. This includes arithmetic operations ($+$, $-$, $*$, $/$, modulo, remainder), bitwise operations ($\neg$, $\lor$, $\land$, bitshifts), concatenation of bitvectors, and extraction of sub-bitvectors.

2.2.5 Effectively Propositional Logic

Effectively Propositional Logic (EPR) [153], also known as Bernays-Schönfinkel class, is a subset of first-order logic that contains formulas of the form $\exists \overline{x} : \forall \overline{y} : F$, where $\overline{x}$ and $\overline{y}$ are disjoint vectors of variables ranging over domain $\mathbb{D}$, and F is a function-free first-order CNF. The formula F can contain predicates over $\overline{x}$ and $\overline{y}$, though.

2.3 Decision Procedures and Reasoning Engines

In the following, we will discuss decision procedures and reasoning engines for the logics introduced in the previous section from a user's perspective.

2.3.1 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [45] are a graph-based representation for formulas in propositional logic. The graphs are rooted and acyclic. There are two terminal nodes, which we denote by $\boxed{0}$ and $\boxed{1}$. Non-terminal nodes are labeled by a variable, have exactly two outgoing edges, and act as decisions: when traversing the graph from the root node, depending on the truth value of the variable labelling a node, one of the outgoing edges is taken. If the terminal node $\boxed{0}$ is reached during such a traversal, then

[1] Technically, defining constants for 0 and 1 is enough if $+$ and $-$ are available.

this means that the formula evaluates to false for this assignment. If $\boxed{1}$ is reached, the formula evaluates to true.

Example 2. A BDD for the formula $f = (x \vee y) \wedge \neg z$ is shown in Figure 2.1. The root node, representing f, is marked with an incoming arrow. Non-terminal nodes are drawn as circles. The solid outgoing edge is taken if the variable written in the node is true, the dashed edge is taken if it is false. The two terminal nodes are drawn as boxes. The graph can be read as follows: If $z =$ true, the entire formula f is false. Otherwise, x is considered. If x is true (and $z =$ false), the formula is true. Otherwise, y is considered. If $y =$ true (and $z = x =$ false), then f is true. If $y =$ false (and $z = x =$ false), then f is false.

Orderdness and Reducedness. BDDs are *ordered* in the sense that for all paths from the root to the terminal nodes, decisions on the variables are always taken in the same order. We will refer to this order as the *variable order* of the BDD. For instance, the variable order in Figure 2.1 is z, x, y. Furthermore, BDDs are *reduced* in the sense that redundant vertices (where the true- and the false-successor are the same node) and isomorphic subgraphs have been eliminated. This reduction serves two purposes. First, it reduces the size of the BDDs. Second, for a fixed variable order, it makes BDDs a *canonical* representation of a propositional formula.

Canonicity. A BDD is a *canonical* representation of a propositional formula in the sense that for a fixed variable order, the same formula will always be represented by isomorphic graphs. This property makes equivalence checks between propositional formulas simple: once the BDDs have been built, all that needs to be done is to compare the graphs. In particular, a satisfiability check can be performed by comparing the BDD with that for false (which has the terminal node $\boxed{0}$ as its root). BDD libraries are usually implemented in such a way that multiple formulas are represented by a single graph with several root nodes [160]. If two formulas are equivalent, they are represented by the same node in the graph. This saves memory (because common subgraphs are stored only once) and allows for equivalence checks between formulas in constant time: all that needs to be done is to check if the root nodes are identical.

Variable (re)ordering. In practice, the size of a BDD crucially depends on the variable ordering that is imposed. For example, a certain sum-of-products formula [45] can be represented with a linear number of nodes in the best ordering, and with an exponential number of nodes in the worst ordering. Unfortunately, it can be shown [192] that the problem of computing a variable ordering that results in at most k times the BDD nodes of the optimal ordering is NP-complete. That is, finding a good variable ordering is a computationally hard problem. As a consequence, BDD libraries mostly rely on heuristics. Particularly important are dynamic reordering heuristics [187], which try to reduce the BDD size automatically while constructing and manipulating BDDs. Additionally (or alternatively), the user of a BDD library can also trigger reorderings with specified heuristics manually.

Variable reordering heuristics are certainly effective in improving the scalability of BDDs, especially in industrial applications such as formal verification of hardware circuits [187]. However, there exist formulas for which no variable ordering yields a small BDD. Even worse, such characteristics cannot only be observed on artificial examples, but also on structures that occur frequently in industrial applications. For instance, for an n-bit multiplier, it can be shown [45] that at least one of the output functions requires at least $2^{n/8}$ BDD nodes for any variable ordering. Together with the recent progress in efficient SAT solving (see below), these scalability issues are among the reasons why BDDs are increasingly displaced in applications like model checking.

Operations on BDDs. BDD libraries like CUDD [198] provide a rich set of operations. Besides the basic Boolean connectives $\neg$, $\vee$, $\wedge$, etc., they offer universal and existential quantification of variables. Hence, BDDs can also be used to reason about Quantified Boolean Formulas (QBFs). Other useful operations are the computation of positive and negative cofactors, as well as swapping of variables in the formula. Satisfying assignments can be computed by traversing some path from the root to the terminal node $\boxed{1}$. BDD libraries often also provide combined operations that can be computed more efficiently than performing the operations in isolation. One example of such a combined operation is $\exists \overline{x} : F_1(\overline{x}, \overline{y}) \wedge F_2(\overline{x}, \overline{z})$, i.e., conjunction followed by existential quantification of some variables. Because of this rich set of operations, it is often not difficult to realize symbolic algorithms (we will introduce this term in Section 2.4) using BDDs as the underlying reasoning engine.

2.3.2 SAT solvers

A SAT solver can decide whether a given propositional formula in CNF is satisfiable. This problem is NP-complete, which means that given solutions can be checked in polynomial time, but no polynomial algorithms to compute solutions are known[2]. Despite this relatively high complexity[3] there have been enormous scalability improvements over the last decades. Today, modern SAT solvers can handle industrial problem instances with millions of variables and clauses [133].

Working principle. Modern SAT solvers [133] are based on the concept of *Conflict-Driven Clause Learning (CDCL)*, where partial assignments that falsify the formula are eliminated by adding a blocking clause to forbid the partial assignment. The current assignment in the search is not just negated to obtain the clause. Instead, a conflict graph is analyzed with the goal of eliminating irrelevant variables and thus learning smaller blocking clauses. This idea is combined with aggressive (so-called *non-chronological*) backtracking to continue the search. This general principle was introduced in 1996 with the SAT solvers GRASP [193]. Modern solvers still follow the same principle [133], but extended with clever data structures for constraint propagation, heuristics to choose variable assignments, restarts of the search, and other improvements. We refer to [25] for more details on these techniques.

SAT competition. One driving force for research in efficient SAT solving is the annual SAT competition[4] [122], held since 2002. It also defines a simple textual format for CNFs, which is called DIMACS [177] and supported by virtually all SAT solvers. A comparison [122] of the best solvers from 2002 to 2011 shows that the number of benchmark instances (of the 2009 benchmark set) solved within 1200 seconds increased from around 50 to more than 170 during this time span. Conversely, the maximum solving time for the 50 simplest benchmarks dropped from around 1100 seconds to around 10 seconds. The plot in [122] summarizing this data does not show any signs of saturation over the years. Hence, further performance improvements can also be expected for the coming years. The SAT solver based synthesis methods presented in Chapter 3 will directly benefit from such improvements.

Solver Features and Notation in this Thesis

In the algorithms presented in this thesis, we will denote a call to a SAT solver by

$$\mathsf{sat} := \textsc{PropSat}\big(F(\overline{x})\big),$$

where $F(\overline{x})$ is a propositional formula in CNF. The variable sat is assigned true if $F(\overline{x})$ is satisfiable, and false otherwise.

Satisfying assignments. Modern SAT solvers do not only decide satisfiability, but can also compute a satisfying assignment for the variables in the formula. We will write

$$(\mathsf{sat}, \mathbf{x}, \mathbf{y}, \ldots) := \textsc{PropSatModel}\big(F(\overline{x}, \overline{y}, \ldots)\big)$$

to denote a call to the solver where we also extract a satisfying assignment in the form of cubes $\mathbf{x}, \mathbf{y}, \ldots$ over the variables $\overline{x}, \overline{y}, \ldots$ occurring in the formula F. The cubes may be incomplete if the value of the missing variables is irrelevant for F to be true. The returned cubes are meaningless if sat is false.

Unsatisfiable cores. Another feature of modern SAT solvers is the efficient computation of unsatisfiable cores, as defined in Section 2.2.1. Given that $\mathbf{x} \wedge F(\overline{x}, \overline{y})$ is unsatisfiable, we will write

$$\mathbf{x}' := \textsc{PropUnsatCore}\big(\mathbf{x}, F(\overline{x}, \overline{y})\big)$$

[2] Even more, if P≠NP, which is widely believed but not proven, no polynomial algorithm exists.

[3] Well, in comparison to the complexities that have to be dealt with in synthesis it is actually not so high.

[4] http://www.satcompetition.org/ (last visit on 2015-08-01).

to denote the extraction of an unsatisfiable core $\mathbf{x}' \subseteq \mathbf{x}$ such that $\mathbf{x}' \wedge F(\overline{x}, \overline{y})$ is still unsatisfiable. Natively, SAT solvers usually compute unsatisfiable cores that are not necessarily minimal. However, a computed core can easily be minimized by trying to drop literals of $\mathbf{x}'$ one by one and checking if unsatisfiability is still preserved. We will denote the computation of a minimal unsatisfiable core by

$$\mathbf{x}' := \textsc{PropMinUnsatCore}\big(\mathbf{x}, F(\overline{x}, \overline{y})\big).$$

In our algorithms, we will use unsatisfiable core computations mostly to generalize discovered facts. In our experience, good generalizations (in the form of small cores) are usually more beneficial than fast ones. Thus, we will usually compute minimal unsatisfiable cores in our algorithms.

Interpolation. Given two CNFs $A(\overline{x}, \overline{y})$ and $B(\overline{x}, \overline{z})$ with $A \wedge B = \mathsf{false}$, we denote the computation of a Craig interpolant $I(\overline{x})$ (such that $A \rightarrow I \rightarrow \neg B$; cf. Section 2.2.1) by

$$I := \textsc{Interpol}(A, B).$$

While SAT solvers usually cannot compute interpolants natively, many of them can output unsatisfiability proofs. An interpolant can then be computed from such an unsatisfiability proof for $A \wedge B$ using different methods [76].

Incremental solving. Modern CDCL-based SAT solvers can solve sequences of similar CNF queries more efficiently than by processing the queries in isolation. For instance, if clauses are only added but not removed between satisfiability checks, all the clauses learned so far can be retained and do not have to be rediscovered again and again. Removing clauses is more problematic. Certain learned clauses may become invalid and need to be removed as well. Clause removals are supported by different solvers in different ways (or not at all). One wide-spread approach is to provide an interface for pushing the current state of the solver onto a stack and restoring it later. A related feature that is supported by many SAT solvers is *assumption literals*, which can be asserted temporarily. In the algorithms presented in this thesis, we will mostly avoid removing clauses from incremental SAT sessions and use assumption literals to enable or disable parts of a formula instead. In this context, will also refer to variables that are introduced for the purpose of enabling or disabling formula parts as *activation variables*.

In general, we will present our synthesis algorithms in a non-incremental way and discuss the use of incremental solving separately. This way, we do not have to introduce notation for adding clauses, resetting the state of a solver, etc., which increases the readability of the algorithms.

2.3.3 QBF Solvers

A QBF solver can decide whether a given Quantified Boolean Formula in PCNF is satisfiable. This problem is PSPACE-complete [138], which means that solving it requires a polynomial amount of memory. However, no NP-time algorithms are known[5]. Hence, from a complexity point of view, QBF problems are (likely to be) strictly harder than SAT problems.

Working principle. While most modern SAT solvers follow the concept of CDCL, the set of techniques applied for QBF solving is more diverse. For instance, the solver DepQBF [155] uses a search-based algorithm (called QDPLL) with conflict-driven clause learning (similar to CDCL SAT solvers) and solution-driven cube learning. The solver Quantor [21] uses variable elimination in order to transform the problem into a purely propositional formula. The solver RAReQS [120] follows the idea of counterexample-guided refinement of solution candidates, where plain SAT solvers are used to compute solution candidates as well as to refute and refine them. None of these techniques is clearly superior — different techniques appear to work well on different benchmarks.

Preprocessing. An important topic in QBF solving is preprocessing. A QBF preprocessor simplifies a QBF before the actual solver is called. It is also possible that the preprocessor solves a QBF problem directly, or reduces it to a propositional formula, for which a SAT solver can be used. Bloqqer [26] is

[5] And it is widely believed, but not proven, that no such algorithms exist.

an example of a modern QBF preprocessor implementing many techniques. It has been shown to have a very positive impact on the performance of different kinds of solvers [26]: when using Bloqqer, the QBF solvers DepQBF [155], Quantor [21], QuBE [98] and Nenofex [154] can solve between 20 % and 40 % more benchmarks (of the benchmark set from the QBFEVAL 2010 competition within 900 seconds). The median execution time decreases by up to a factor of 50 (achieved for QuBE) due to Bloqqer [26].

Competitions. Similar to SAT solving, there are also competitions in QBF solving (QBFEVAL[6] and the QBF Gallery[7]) with the aim of collecting benchmarks as well as assessing and advancing the state of the art in QBF research and tool development. The input format for these competitions is called QDIMACS, and is essentially just an extension of the DIMACS format with a quantifier prefix. While the QBF competitions definitely witness solid progress in scalability over the years, it seems that QBF has not yet reached the maturity of SAT, especially when it comes to industrial applications such as formal verification, where the scalability is often insufficient [16]. However, because QBF is a much younger research field than SAT, future scalability improvements may be even more significant. The QBF-based synthesis algorithms presented in this thesis would directly benefit from such developments.

Solver Features and Notation in this Thesis

Similar to our notation for SAT solvers, we will write

$$\mathsf{sat} := \textsc{QbfSat}\big(Q_1\overline{x} : Q_2\overline{y} : \dots F(\overline{x}, \overline{y}, \dots)\big)$$

to denote a call to a QBF solver, where F is a propositional formula in CNF, and $Q_i \in \{\exists, \forall\}$. As before, sat will be assigned true if the QBF is satisfiable and false otherwise.

Satisfying assignments. Many existing QBF solvers cannot only decide the satisfiability of formulas, but also compute satisfying assignments for variables that are quantified existentially on the outermost level. We will write

$$(\mathsf{sat}, \mathbf{a}, \mathbf{b} \dots) := \textsc{QbfSatModel}\big(\exists\overline{a} : \exists\overline{b} : \dots Q_1\overline{x} : Q_2\overline{y} : \dots F(\overline{a}, \overline{b}, \dots, \overline{x}, \overline{y}, \dots)\big)$$

to denote the extraction of such a satisfying assignment in the form of cubes $\mathbf{a}, \mathbf{b}, \dots$ over the variable vectors $\overline{a}, \overline{b}, \dots$ quantified existentially on the outside. In general, satisfying assignments cannot be extracted when applying QBF preprocessing, because preprocessing techniques are often not model preserving. To remedy this situation, we extended the popular QBF preprocessors Bloqqer to preserve satisfying assignments [189] so that we can use QBF preprocessing in synthesis algorithms that require satisfying assignments. However, this work will not be presented in detail in this thesis, and is thus only mentioned briefly at this point.

Unsatisfiable cores. Certain QBF solvers, such as DepQBF [156], can also compute unsatisfiable cores natively. However, this feature cannot be used with preprocessing straightforwardly. Furthermore, we did not encounter significant performance improvements in our experiments compared to minimizing the core in an explicit loop. Hence, we do not introduce dedicated notation for unsatisfiable QBF cores and use explicit minimization loops in our algorithms instead.

Incremental solving. Comprehensive approaches for incremental QBF solving have only been proposed very recently [156]. However, incremental solving cannot yet be used in combination with QBF preprocessing, because existing preprocessors are inherently non-incremental. We experimented with incremental solving in our synthesis algorithms. However, for many cases, preprocessing turned out to much more beneficial than incremental solving. We will therefore refrain from introducing notation for incremental QBF solving, and discuss possibilities for incremental solving separately.

[6] `http://www.qbflib.org/index_eval.php` (last visit on 2015-08-01).

[7] `http://qbf.satisfiability.org/gallery/` (last visit on 2015-08-01).

2.3.4 First-Order Theorem Provers

First-order logic is undecidable [115], that is, an algorithm to decide the satisfiability (or validity) of every possible first-order logic formula cannot exist. Yet, incomplete algorithms and tools *do* exist, and they perform well on many practical problems. Similar to SAT and QBF, there is also a competition for automatic theorem provers to solve problems in first-order logic and subsets thereof. It is called CASC[8] [204] and exists since 1996. Benchmarks for the competition are taken from the TPTP library [203], which defines a common format for first-order logic problems.

In this thesis, we are not so much interested in full first-order logic, but rather in the subset called Effectively Propositional Logic (EPR). In contrast to full first-order logic, EPR is actually decidable [153] (the problem is NEXPTIME-complete). The CASC competition also features a track for EPR. From 2008 to 2014, this track was always won by iProver [146]. iProver is an instantiation-based solver and can thus not only decide the satisfiability of EPR formulas, but also compute models in form of concrete realizations for the predicates. This feature makes iProver particularly suitable for synthesis.

2.3.5 SMT Solvers

A Satisfiability Modulo Theories (SMT) solver can be seen as a first-order theorem prover that is specialized towards deciding the satisfiability of formulas with respect to some background theories. Many SMT solvers can only handle quantifier-free formulas, and can thus also be understood as extensions of SAT solvers to more expressive (but typically decidable) logics. This second view also explains the nature of the most common algorithms implemented in SMT solvers: algorithms for propositional satisfiability are combined with theory-specific reasoning engines [169]. This approach is usually much more efficient than using a first-order theorem prover with axioms for the theory[9], especially for realistic applications [180].

Working principle. There are several general strategies for SMT solving [14]. *Eager encoding* schemes attempt to transform the problem into an equisatisfiable propositional formula by eagerly instantiating all consequences of the theory axioms on the formula. The propositional formula is then solved by a SAT solver. Of course, this can significantly blow up the formula size. However, due to the availability of powerful SAT solvers, this approach can nevertheless be efficient. For bitvector arithmetic, this approach is called *bit-blasting* and implemented in efficient solvers such as Boolector [44]. *Lazy encoding* is an approach where a SAT solver and a theory solver for conjunctive statements interact. In its simplest form, the SAT solver first computes a satisfying assignment for the propositional skeleton of the formula, i.e., a truth value for all predicate occurrences, such that the formula becomes true. In case of unsatisfiability, the entire formula is unsatisfiable. In case of satisfiability, the theory solver checks whether the computed truth assignment for the predicate occurrences is feasible in the theory. If so, the entire formula is satisfiable modulo the theory. Otherwise, a blocking clause is computed, which prevents the SAT solver from producing the same assignment (as well as other assignments that are inconsistent in the theory) again. DPLL(T) is a variant of this lazy encoding approach where the SAT solver and the theory solver are more tightly integrated. We refer to [14] for a more elaborate discussion of SMT solving techniques.

Competitions. Since 2005, there has been a yearly competition for SMT solvers, called SMT-COMP[10]. It stimulates research and tool development. It also defines a standard input format for SMT problems, which is called SMT-LIB (version 2) [13].

[8] http://www.cs.miami.edu/~tptp/CASC/ (last visit on 2015-08-01).

[9] This is not even possible in some cases because some theories cannot be captured by a finite set of first-order axioms [180].

[10] http://www.smtcomp.org/ (last visit on 2015-08-01).

Solver Features and Notation in this Thesis

Similar to SAT solvers, SMT solvers can compute satisfying assignments, unsatisfiable cores, and perform incremental solving. We will write

$$\mathsf{sat} := \textsc{SmtSat}\big(F(\overline{x})\big)$$

to denote a call to an SMT solver, where F is a formula over some vector of (potentially non-Boolean) variables $\overline{x} = (x_1, \ldots, x_n)$. As before, sat is assigned true if the formula F is satisfiable modulo the used theories, and false otherwise.

Satisfying assignments. We will write

$$(\mathsf{sat}, \mathbf{x}, \mathbf{y}, \ldots) := \textsc{SmtSatModel}\big(F(\overline{x}, \overline{y}, \ldots)\big)$$

to denote a call to the solver where we also extract a satisfying assignment. Recall that satisfying assignments for propositional formulas were represented by cubes. In the case of SMT, a satisfying assignment is simply a vector of constants, one for each variable.

Unsatisfiable cores. Let $F(\overline{x}, \overline{y}, \ldots)$ be a formula in which the variables in $\overline{x}$ are all Boolean and let $\mathbf{x}$ be a cube over $\overline{x}$. Given that $\mathbf{x} \wedge F(\overline{x}, \overline{y}, \ldots)$ is unsatisfiable, we write

$$\mathbf{x}' := \textsc{SmtUnsatCore}\big(\mathbf{x}, F(\overline{x}, \overline{y}, \ldots)\big)$$

to denote the extraction of an unsatisfiable core $\mathbf{x}' \subseteq \mathbf{x}$ such that $\mathbf{x}' \wedge F(\overline{x}, \overline{y}, \ldots)$ is still unsatisfiable. The computation of a minimal unsatisfiable core will be denoted by

$$\mathbf{x}' := \textsc{SmtMinUnsatCore}\big(\mathbf{x}, F(\overline{x}, \overline{y}, \ldots)\big).$$

Incremental solving. Just as for SAT solvers and QBF solvers, we do not introduce notation for incremental SMT solving, but discuss possibilities for incremental solving separately.

2.4 Symbolic Encoding and Symbolic Computations

Formal methods for verification or synthesis must be able to deal with large sets of states or large sets of possible inputs efficiently. *Symbolic encoding* [115, page 383] is a way to represent large sets of elements compactly using formulas. Set elements are represented by assignments to a set of variables. Formulas over these variables characterize which elements are contained in a set: if the formula evaluates to true for a particular variable assignment, then the corresponding element is part of the set. If the formula evaluates to false, then the element is not contained. Such a formula is called the *characteristic formula* of the set.

Example 3. Consider the set A of all integers from 0 to 65535. We can use 16 Boolean variables $\overline{x} = (x_0, \ldots, x_{15})$ to encode subsets of A symbolically. The variables represent the bits of the binary encoding of a number, with x_0 being the least significant bit. An explicit representation of the set $A_0 = \{0, 2, 4, \ldots, 65534\}$ of all even numbers would have to enumerate 32768 elements. In a symbolic representation, the set of even numbers can be represented by the propositional formula $F_0(\overline{x}) = \neg x_0$, requiring that the least significant bit is false and all other bits are arbitrary. The set $A_1 = \{49152, 49153, \ldots, 65535\}$ of all numbers greater or equal to 49152 can be represented symbolically using the formula $F_1(\overline{x}) = x_{15} \wedge x_{14}$, stating that the two most significant bits must be set.

Characteristic formulas cannot only be used to *represent* sets. We can also perform set operations directly on the formulas. A set union $A_0 \cup A_1$ can be realized as disjunction of the corresponding characteristic formulas F_0 and F_1, intersection corresponds to conjunction, and a complement to the negation of the characteristic formula. The formula false represents the empty set, the formula true represents the set of all elements in the domain.

Example 4. Continuing Example 3, the set $A_0 \cap A_1$ of even numbers greater or equal to 49152 can be computed symbolically as $F_0(\overline{x}) \wedge F_1(\overline{x}) = \neg x_0 \wedge x_{15} \wedge x_{14}$. The set $A_1 \setminus A_0$ of odd numbers greater or equal to 49152 can be computed symbolically as $F_1(\overline{x}) \wedge \neg F_0(\overline{x}) = x_{15} \wedge x_{14} \wedge x_0$.

In this thesis, we will often handle sets and their symbolic representations interchangeably. For instance, we may say "the set of states $F(\overline{x})$" although F is a formula over state variables $\overline{x}$, representing the set symbolically.

2.5 Hardware Synthesis from Safety Specifications

In this section, we define the hardware controller synthesis problem from safety specifications and introduce the relevant concepts from game theory to solve the problem. We also present a standard textbook solution. It will serve as the starting point for the SAT-based methods that will be presented in Section 3. The last subsection then briefly discusses other common specification formats, which will mostly be relevant for understanding our discussion of related work as well as suggestions for future work.

2.5.1 Safety Specifications

Intuitively, a safety specification expresses that certain "bad things" can never happen in a system. This stands in contrast to liveness properties, which stipulate that certain "good things" must happen eventually. In this light, our focus on synthesis procedures for safety specifications may appear rather restrictive at the first glance. However, synthesis algorithms for safety specifications can be useful even for specifications that contain liveness properties. First, bounded synthesis approaches [81, 86] can reduce synthesis from richer specifications to pure safety synthesis problems. This is often done by setting a bound on the reaction time. For instance, instead of requiring that some event happens eventually, one may require that it happens within at most k steps. Clearly, a realization of the latter is also a realization of the former. This approach [86] has been followed in the SyntComp competition for translating LTL benchmarks into safety specifications automatically [117], but can also be applied when writing a safety specification manually. By choosing k as low as possible (such that the problem is still realizable), we may even get better systems in the sense that they react faster. A second reason for the importance of safety synthesis procedures is that safety properties often make up the bulk of a specification and they can be handled in a compositional manner: the safety synthesis problem can be solved before the other properties are handled [195].

Safety specifications in SyntComp. The SyntComp [117] synthesis competition defines safety specification benchmarks as hardware circuits in AIGER[11] format, as illustrated in Figure 2.2. The circuits have uncontrollable inputs $\overline{i}$, controllable inputs $\overline{c}$, flip-flops to store a number of state bits $\overline{x}$, and one output "error" signaling specification violations. The corresponding synthesis problem is to construct a circuit that defines the controllable inputs $\overline{c}$ based on the uncontrollable inputs $\overline{i}$ and the state $\overline{x}$ in such a way that the error output can never reach the value true. This unknown circuit to be constructed is denoted with a question mark in Figure 2.2. We will also refer to the controllable inputs as *control signals* to emphasize that these signals are not intended to be inputs of the final system.

The specification illustrated in Figure 2.2. can be seen as a runtime monitor, declaratively encoding the design intent for the system to be synthesized. Another view is that the specification is a plant which needs to be controlled, or a sketch of a hardware circuit where the implementation for certain signals is still missing. Hence, this format nicely fits the concept of controller synthesis, allowing for a mixed imperative/declarative programming paradigm. Formally, we define a safety specification as follows.

Definition 5 (Safety Specification)**.** *A safety specification is a tuple $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T, P)$, where*

- *$\overline{x}$ is a vector of Boolean state variables,*

[11] http://fmv.jku.at/aiger/ (last visit on 2015-08-01).

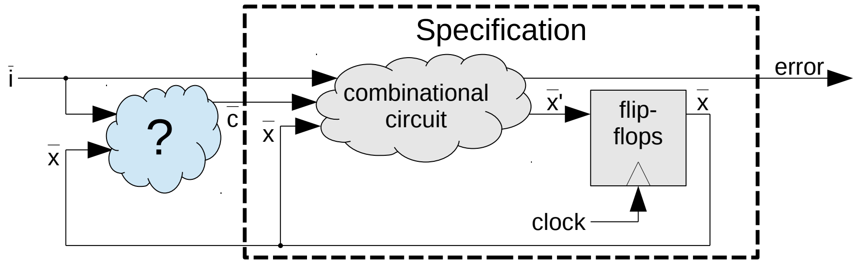


Figure 2.2: Circuit representation of a safety specification. The variable vector $\overline{i}$ represents uncontrollable inputs, $\overline{c}$ represents controllable inputs, and $\overline{x}$ is a vector of state bits. The system to be synthesized is marked with a question mark. It must define the signals $\overline{c}$ such that the error output can never become true.

- *$\overline{i}$ is a vector of uncontrollable, Boolean input variables,*
- *$\overline{c}$ is a vector of controllable, Boolean input variables,*
- *$I(\overline{x})$ is an initial condition, expressed as a propositional formula over the state variables,*
- *$T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is a transition relation, expressed as a propositional formula over the variables $\overline{x}$, $\overline{i}$, $\overline{c}$, and $\overline{x}'$, where $\overline{x}'$ denotes the next-state copy of $\overline{x}$,*
- *the transition relation $T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is complete in the sense that $\forall \overline{x}, \overline{i}, \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$,*
- *T is deterministic in that $\forall \overline{x}, \overline{i}, \overline{c}, \overline{x}'_1, \overline{x}'_2 : \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'_1) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'_2)\big) \rightarrow (\overline{x}'_1 = \overline{x}'_2)$, and*
- *$P(\overline{x})$ is a propositional formula representing the set of safe states in $\mathcal{S}$.*

A *state* of $\mathcal{S}$ is an assignment to all state variables $\overline{x}$. We will represent such assignments (and thus states of $\mathcal{S}$) as $\overline{x}$-minterms $\mathbf{x}$. In the spirit of symbolic encoding as introduced Section 2.4, a formula $F(\overline{x})$ over the state variables $\overline{x}$ represents the set of all states $\mathbf{x}$ for which $\mathbf{x} \models F(\overline{x})$ holds. In this way, the formula $I(\overline{x})$ defines a set of initial states, and the formula $P(\overline{x})$ defines the set of safe states. Similarly, the formula T defines allowed state transitions: a transition from the current state $\mathbf{x}$ to the next state $\mathbf{x}'$ is allowed with input $\mathbf{i}$ and $\mathbf{c}$ if and only if $\mathbf{x} \wedge \mathbf{i} \wedge \mathbf{c} \wedge \mathbf{x}' \models T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$. Definition 5 requires that the transition relation T is both deterministic and complete. That is, for any state $\mathbf{x}$ and input $\mathbf{i}, \mathbf{c}$, the next state $\mathbf{x}'$ is always uniquely defined.

2.5.2 Safety Games

A specification $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T, P)$ can be seen as a game between two players: the *environment* and the *system* we wish to synthesize (see also Figure 1.2). Depending on the context, we will thus refer to $\mathcal{S}$ either as a specification or as a game.

Plays. The game starts in one of the initial states (chosen by the environment), and is played in rounds. In every round j, the environment first chooses an assignment $\mathbf{i}_j$ to the uncontrollable inputs $\overline{i}$. Next, the system picks an assignment $\mathbf{c}_j$ to the controllable inputs $\overline{c}$. The transition relation T then computes the next state $\mathbf{x}_{j+1}$. This is repeated indefinitely. The resulting sequence $\mathbf{x}_0, \mathbf{x}_1 \ldots$ of states is called a *play*. Formally, we have that $\mathbf{x}_0 \models I(\overline{x})$ and $\mathbf{x}_j \wedge \mathbf{x}'_{j+1} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is satisfiable (with some $\mathbf{i}_j$ and $\mathbf{c}_j$ chosen by the players) for all $j \geq 0$. A play $\mathbf{x}_0, \mathbf{x}_1 \ldots$ is *won* by the system and lost by the environment if $\forall j : \mathbf{x}_j \models P(\overline{x})$, i.e., if only safe states are visited. Otherwise, the play is *lost* by the system and won by the environment.

Preimages. Let $F(\overline{x})$ be a formula representing a certain set of states. The mixed preimage

$$\mathsf{Force}^s_1\big(F(\overline{x})\big) = \forall \overline{i} : \exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

represents all states from which the system can enforce that some state of F is reached in exactly one step. Analogously,

$$\mathsf{Force}^e_1\big(F(\overline{x})\big) = \exists \overline{i} : \forall \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

gives all states from which the environment can enforce that F is visited in one step. We also define the cooperative preimage

$$\mathsf{Reach}_1\big(F(\overline{x})\big) = \exists \overline{i}, \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

denoting the set of all states from which F can be reached cooperatively by the two players. The following dualities can easily be shown:

- $\neg\mathsf{Force}^s_1(F) = \mathsf{Force}^e_1(\neg F)$ holds because, intuitively, the states from which the system cannot enforce that F is reached must be the states from which the environment can enforce that $\neg F$ is reached.
- $\neg\mathsf{Force}^e_1(F) = \mathsf{Force}^s_1(\neg F)$ holds because, dually, the states from which the environment cannot enforce that F is reached must be the states from which the system can enforce that $\neg F$ is reached.

Furthermore, we have that $\mathsf{Reach}_1(F_1) \vee \mathsf{Reach}_1(F_2) = \mathsf{Reach}_1(F_1 \vee F_2)$. Yet, the following equivalence does **not** hold in general: $\mathsf{Force}^s_1(F_1) \vee \mathsf{Force}^s_1(F_2) \not\leftarrow \mathsf{Force}^s_1(F_1 \vee F_2)$. The intuitive reason is that there may be states from which the environment controls whether F_1 or F_2 is visited next, and the system can only ensure that one of the two regions is reached. Such states would falsify $\mathsf{Force}^s_1(F_1 \vee F_2) \rightarrow \mathsf{Force}^s_1(F_1) \vee \mathsf{Force}^s_1(F_2)$. This difference in compositionality between Reach_1 and Force^s_1 explains why some ideas from verification often cannot be ported to synthesis straightforwardly.

Strategies. In this work, we focus on memoryless strategies because these strategies are sufficient[12] for safety games [205]. A (memoryless) *strategy* for the system player in the game $\mathcal{S}$ is a formula $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ that specializes T in the sense that

- $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \rightarrow T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ and
- $\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$.

The implication in the first bullet requires that the strategy may only allow state transitions that are also allowed by the transition relation. The second bullet expresses that the strategy must be complete with respect to the current state and uncontrollable input: for every state $\mathbf{x}$ and input $\mathbf{i}$, the strategy must contain some way for the system to choose $\mathbf{c}$ (and some next state, but the next state is uniquely defined by T already). For a particular situation, the strategy can allow many possibilities to choose $\mathbf{c}$, though. A strategy for the system is *winning* if all plays that can be constructed by following S instead of T are won by the system. The *winning region* $W(\overline{x})$ is the set of all states from which a winning strategy exists. That is, if the play would start in some arbitrary state of the winning region, the system player would have a strategy to always win the game.

System implementations. A *system implementation* is a function $f : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow 2^{\overline{c}}$ to uniquely define the control signals $\overline{c}$ based on the current state and the uncontrollable inputs $\overline{i}$. A system implementation f *implements* a strategy S if $\forall \overline{x}, \overline{i} : \exists \overline{x}' : S\big(\overline{x}, \overline{i}, f(\overline{x}, \overline{i}), \overline{x}'\big)$, that is, if for every state $\mathbf{x}$ and input $\mathbf{i}$, the control value $\mathbf{c} = f(\mathbf{x}, \mathbf{i})$ computed by f is allowed by the strategy S. A system implementation f *realizes* a safety specification $\mathcal{S} = \big(\overline{x}, \overline{i}, \overline{c}, I(\overline{x}), T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'), P(\overline{x})\big)$ if all plays of $\mathcal{S}' = \big(\overline{x}, \overline{i}, \emptyset, I(\overline{x}), T(\overline{x}, \overline{i}, f(\overline{x}, \overline{i}), \overline{x}'), P(\overline{x})\big)$ are won by the system player, i.e., visit only safe states. Here, $\mathcal{S}'$ is a simplified version of the game $\mathcal{S}$ where the moves of the system player are already defined by f, i.e., the system player has no choices left. A safety specification is *realizable* if a system implementation that realizes it exists. Given a winning strategy S for a safety specification $\mathcal{S}$, every implementation f of the winning strategy S realizes the specification $\mathcal{S}$. This follows directly from the definition of the winning strategy. Hence, a system implementation for a given safety specification $\mathcal{S}$ can be constructed by computing a winning strategy S for $\mathcal{S}$ and then computing an implementation f of S.

2.5.3 Synthesis Algorithms for Safety Specifications

Given an explicit representation of the safety specification $\mathcal{S}$ as a game graph (with vertices representing states and edges representing state transition) the problem of deciding the realizability of a safety speci-

[12] Memoryless strategies are sufficient in the sense that, if a strategy to win the game exists, then there also exists a memoryless strategy to win the game.

Algorithm 2.1 SafeWin: A textbook algorithm for computing a winning region in a safety game.

```
 1: procedure SafeWin$\big((\overline{x}, \overline{i}, \overline{c}, I, T, P)\big)$, returns: The winning region $W$ or false
 2:     $F := P$
 3:     while $F$ changes do
 4:         $F := F \wedge \mathsf{Force}^s_1(F)$
 5:         if $I \not\rightarrow F$ then
 6:             return false
 7:         end if
 8:     end while
 9:     return $F$
10: end procedure
```

fication is known to be solvable in linear time [205]. When starting from our symbolic representation $\mathcal{S}$, the realizability problem is EXP-time complete [171].

A *synthesis algorithm* for safety specifications takes as input a safety specification $\mathcal{S}$ and computes a system implementation realizing this specification if such an implementation exists. If no such implementation exists, the algorithm reports unrealizability. Wolfgang Thomas [205] sketches the standard textbook algorithm for solving this problem. It proceeds in two steps. First, a winning strategy is computed. Second, the winning strategy is implemented in a circuit. This process is elaborated in the following two subsections.

2.5.3.1 Computing a Winning Strategy

The computation of a winning strategy $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ for the game $\mathcal{S} = \big(\overline{x}, \overline{i}, \overline{c}, I(\overline{x}), T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'), P(\overline{x})\big)$ is achieved by computing the winning region $W(\overline{x})$ of the game $\mathcal{S}$ using the procedure SafeWin, shown in Algorithm 2.1. The winning region W is built up in the variable F. Initially, F represents the set of all safe states P. Line 4 retains only those states of F from which the system player can enforce that the play stays in a state of F also in the next step. This operation is repeated as long as the state set F changes. If the set of initial states I is not contained in F any more, the procedure aborts, returning false to signal unrealizability of the specification. Otherwise, the final version of F is returned as the winning region. All operations that are performed in this algorithm can easily be realized using BDDs.

If the specification is realizable, i.e., SafeWin did not return false, a winning strategy S is computed from the winning region W. For safety specifications, S can be defined as

$$S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') = T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(W(\overline{x}) \rightarrow W(\overline{x}')\big).$$

That is, the transition relation must always be respected. Furthermore, if the current state is in the winning region, then the next state must be contained in the winning region as well. This strategy will enforce the specification because $I \rightarrow W$, i.e., all initial states are contained in the winning region (otherwise SafeWin would have signaled unrealizability). When starting from a state of the winning region, the strategy ensures that the next state will be in the winning region again. Finally, the winning region W can only contain safe states, i.e., $W \rightarrow P$. Hence, only safe states can be visited when following the strategy.

2.5.3.2 Computing a System Implementation from a Winning Strategy

The second step is to compute a system implementation that implements the strategy, and to realize this implementation in form of a circuit. This can be done by computing a Skolem function for the variables $\overline{c}$ in the formula

$$\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}, \overline{x}'),$$

Algorithm 2.2 CofSynt: A cofactor-based algorithm for computing an implementation of a strategy.

```
1:  procedure CofSynt$\big(S(\overline{x},\overline{i},\overline{c},\overline{x}')\big)$, returns: $f_1,\ldots,f_n : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow \mathbb{B}$
2:      for $c_j \in \overline{c}$ do
3:          $C_1(\overline{x},\overline{i}) := \exists \overline{x}', \overline{c} : S\big(\overline{x},\overline{i},(c_0,\ldots,c_{j-1},\mathsf{true},c_{j+1},\ldots,c_n),\overline{x}'\big)$
4:          $C_0(\overline{x},\overline{i}) := \exists \overline{x}', \overline{c} : S\big(\overline{x},\overline{i},(c_0,\ldots,c_{j-1},\mathsf{false},c_{j+1},\ldots,c_n),\overline{x}'\big)$
5:          $C(\overline{x},\overline{i}) := \neg C_1(\overline{x},\overline{i}) \vee \neg C_0(\overline{x},\overline{i})$
6:          $F_j(\overline{x},\overline{i}) := \mathsf{simplify}(C_1, C)$
7:          $S(\overline{x},\overline{i},\overline{c},\overline{x}') := S(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(c_j \leftrightarrow F_j(\overline{x},\overline{i})\big)$
8:      end for
9:      return $F_1,\ldots,F_n$
10: end procedure
```

i.e., a function $f : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow 2^{\overline{c}}$ such that

$$\forall \overline{x}, \overline{i} : \exists \overline{x}' : S\big(\overline{x}, \overline{i}, f(\overline{x},\overline{i}), \overline{x}'\big)$$

holds. Usually, we prefer simple functions that can be implemented in small circuits. A survey of existing methods to solve this problem can be found in our FMCAD'12 [82] publication. One widely used method is presented in the following.

The cofactor-based method. The cofactor-based method presented by Bloem et al. [36] can be considered as the "standard method" for computing an implementation from a strategy. It is outlined in Algorithm 2.2. The input is a strategy S, the output is a set of functions $f_1,\ldots,f_n : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow \mathbb{B}$, each one defining one control signal of $\overline{c} = (c_1,\ldots,c_n)$. Together, these functions define $f : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow 2^{\overline{c}}$. The CofSynt procedure computes one f_j after the other. In Line 3, a formula $C_1(\overline{x},\overline{i})$ is constructed. It represents the set of all valuations of $\overline{x}$ and $\overline{i}$ in which $c_j = \mathsf{true}$ is allowed by the strategy. It is computed as the positive cofactor of S with respect to c_j, while all signals that are currently not relevant are quantified existentially. Similarly, Line 4 computes all situations where $c_j = \mathsf{false}$ is allowed by the strategy. Our definition of a strategy implies that $C_1(\overline{x},\overline{i}) \vee C_0(\overline{x},\overline{i}) = \mathsf{true}$, i.e., one of the two values is always allowed (but sometimes both are allowed). Next, Line 5 computes the *care set* C, i.e., the set of all situations in which the output matters. Outside of this care set, the value of c_j can be set arbitrarily. Line 6 uses this information to simplify C_1: The procedure simplify returns some F_j which is equal to C_1 wherever C is true, and arbitrary where C is false. When using BDDs as the underlying reasoning engine, this simplification can be implemented with the BDD operation Restrict [66]. However, this is an optional optimization to obtain smaller circuits. Setting $F_j = C_1$ would work as well. Finally, Line 7 refines the strategy S with the computed implementation for the control signal c_j. This step is necessary because some control signals may depend on others, so fixing the implementation of one control signal may restrict other control signals.

Illustration. Figure 2.3 illustrates one iteration of the CofSynt procedure graphically. The box represents the set of all possible assignments to the variables $\overline{x}$ and $\overline{i}$. The region C_1 contains all situations where $c_j = \mathsf{true}$ is allowed. Similarly, C_0 contains all situations where $c_j = \mathsf{false}$ is allowed. The overlap of the two regions is colored in dark gray. Hence, the dark gray region is the set of situations where both $c_j = \mathsf{true}$ and $c_j = \mathsf{false}$ is allowed. It corresponds to the negation $\neg C$ of the care set C. Note that each point in the box is either contained in C_1 or in C_0 (or in both). The function F_j defining c_j is shown in blue. Outside of the dark gray don't-care area $\neg C$ it matches C_1 precisely. In the don't-care area it can be different, though. These properties are enforced by the procedure simplify, called in Line 6 of CofSynt. Exploiting the freedom in the don't-care region can result in simpler formulas and thus in smaller circuits. In Figure 2.3, this is indicated by F_j being much more regular than C_1.

Computing circuits. In order to obtain an implementation f in form of a hardware circuit, the individual functions f_j, defined as formulas F_j, need to be transformed into a network of gates. In principle, this is not difficult: each F_j is a propositional formula (if quantifiers are left, they can be expanded) and

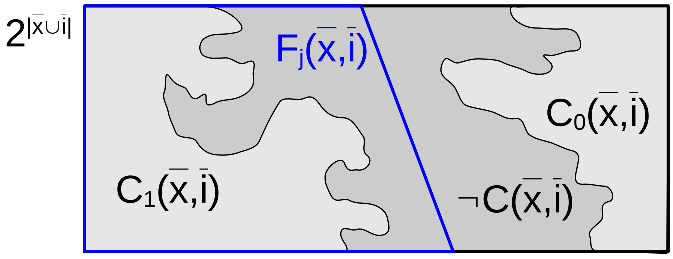


Figure 2.3: Working principle of CofSynt. The box represents the set $2^{|\overline{x} \cup \overline{i}|}$ of all possible assignments to $\overline{x}$ and $\overline{i}$. The dark gray overlap between the regions C_1 and C_0 is the *don't-care* region $\neg C$. The solution F_j matches C_1 save for the don't-care region. Exploiting the freedom in the don't-care region results in a simpler formula F_j.

the structure of the formula can directly be translated into gates. If BDDs are used, the decision graph can directly be translated into a circuit by translating each decision node into a multiplexer.

2.5.4 Other Specification Formats

We briefly sketch some other common specification formats in this subsection. While this is not strictly necessary for understanding the synthesis algorithms presented in this thesis, it is still helpful for understanding the differences to related work.

Reachability specifications. A reachability specification $\mathcal{R} = (\overline{x}, \overline{i}, \overline{c}, I, T, F)$ is defined similar to our safety specification $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T, P)$. The difference lies only in the objective: while $\mathcal{S}$ is satisfied if the safe states $P(\overline{x})$ are never left, $\mathcal{R}$ is satisfied if a certain set $F(\overline{x})$ of target states is reached at least once. There is an obvious duality between safety and reachability: from a certain state $\mathbf{x}$, the system can enforce that the next state is in $P(\overline{x})$ if and only if the environment cannot enforce that the next state is in $\neg P(\overline{x})$. Thus, the winning region of a safety game $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T, P)$ can be computed by first computing the winning region of the reachability game $\mathcal{R} = (\overline{x}, \overline{i}, \overline{c}, I, T, \neg P)$ using Force_1^e instead of Force_1^s, and then taking the complement. Vice versa, the winning region for a reachability game can be computed by solving a safety game.

Büchi specifications. A Büchi specification $\mathcal{B} = (\overline{x}, \overline{i}, \overline{c}, I, T, F)$ is defined just like a safety specification, but the set $F(\overline{x})$ of target states needs to be visited infinitely often instead of only once. A given reachability specification can be transformed into an equivalent Büchi specification by making the target states $F(\overline{x})$ a trap-region that can never be left once reached. Similarly, a safety specification can be transformed into an equivalent Büchi specification by making the unsafe states $\neg P(\overline{x})$ a trap-region and using $P(\overline{x})$ as target states.

Generalized Büchi specifications. A generalized Büchi specification $\mathcal{B}_G = (\overline{x}, \overline{i}, \overline{c}, I, T, \mathcal{F})$ is defined via a set $\mathcal{F} = \{F_1, \ldots F_n\}$ of sets of target states. Each state set $F_i(\overline{x})$ with $1 \leq i \leq n$ needs to be visited infinitely often.

Generalized Reactivity of Rank 1 (GR(1)) [36] specifications. A GR(1) specification essentially consists of two generalized Büchi specifications: one expressing assumptions about the environment, and the other one expressing guarantees about the system. The guarantees need to be satisfied (only) if the assumptions are satisfied. The worst case complexity of deciding realizability for GR(1) specifications is quadratic in the variable space [36] (which is in turn singly exponential in $|\overline{x}| \cdot |\overline{i}| \cdot |\overline{c}|$).

Linear Temporal Logic (LTL) [175] specifications. LTL specifications are defined as formulas over Boolean input- and output variables. Besides the usual Boolean operators $(\neg, \vee, \wedge, \ldots)$ to combine sub-formulas, the following temporal operators are allowed. The operator $\mathsf{X}(\varphi)$, where φ is itself an LTL formula, expresses that φ must hold in the next time step. The operator $\varphi \mathbin{\mathsf{U}} \psi$, where φ and ψ are LTL formulas, requires that φ must be $\mathtt{true}$ until the point in time where ψ becomes $\mathtt{true}$, and ψ must become $\mathtt{true}$ eventually. Based on these two temporal operators, other operators can be defined. The

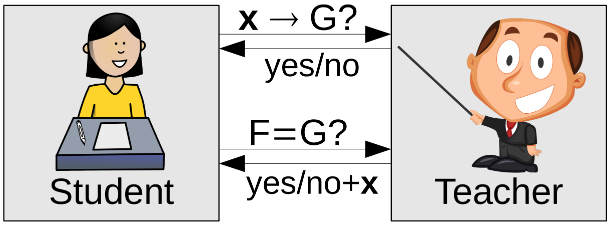


Figure 2.4: Interaction between student and teacher in query learning. Subset queries ask if $\mathbf{x} \rightarrow G(\overline{x})$ and are answered by yes or no. Equivalence queries ask if $F = G$. In case the answer is no, the teacher also provides a counterexample $\mathbf{x}$.

operator $\mathsf{F}(\varphi) = \mathsf{true}\ \mathsf{U}\ \varphi$ expresses that φ must become true at some point in the future. The operator $\mathsf{G}(\varphi) = \neg\,\mathsf{F}(\neg\varphi)$ requires φ to hold in all time steps. As already mentioned in the introduction, synthesis from LTL specifications has a doubly exponential worst case complexity [186].

2.6 Learning by Queries

In this section, we discuss concepts for learning propositional formulas based on queries, as introduced by Angluin [7]. We will use the basic concept of query learning in our SAT-based algorithms for hardware controller synthesis. We mostly follow the terminology of Crama and Hammer [68, Chapter 7] and refer to this book also for a more elaborate discussion.

2.6.1 Basic Concept

The goal of query learning is to compute a small representation F of a propositional formula $G(\overline{x})$ over a given set $\overline{x}$ of Boolean variables. As illustrated in Figure 2.4, this is achieved by two parties in interaction: the *student* (or learner) and the *teacher* (or oracle). The student can ask two kinds of questions:

- A *subset query* asks if a given (potentially incomplete) cube $\mathbf{x}$ is fully contained in $G(\overline{x})$, i.e., if the implication $\mathbf{x} \rightarrow G$ holds. The answer to this question is either yes or no. In algorithms, we will denote such queries by $\mathsf{SUB}(\mathbf{x}, G)$.
- An *equivalence query* asks if a given candidate formula $F(\overline{x})$ is equivalent to $G(\overline{x})$. The answer is again either yes or no. However, in the no-case, the teacher also returns a *counterexample* $\mathbf{x}$ in form of an $\overline{x}$-minterm witnessing the difference. A counterexample is either a *false-positive* with $\mathbf{x} \models F$ and $\mathbf{x} \not\models G$ or a *false-negative* with $\mathbf{x} \not\models F$ and $\mathbf{x} \models G$. In algorithms, we will denote equivalence queries by $\mathsf{EQ}(F, G)$.

A *membership query* is a special form of a subset query where $\mathbf{x}$ is an $\overline{x}$-minterm, i.e., a complete cube.

2.6.2 Learning Algorithms

The general pattern for query learning algorithms is that they start with some initial "guess" of the target function. In a loop, they then perform equivalence queries. If counterexamples are returned, the guess of the target function is refined to eliminate the counterexample. The refinement may involve membership- and subset queries, and distinguishes the algorithms. Concrete algorithms are presented in the following.

Learning a DNF. The procedure DNFLEARN [68, Chapter 7], presented in Algorithm 2.3, computes a DNF representation of a given formula $G(\overline{x})$ using equivalence queries and subset queries. It starts with the initial guess $F = \mathsf{false}$. This guess is then refined based on the counterexamples that are returned by the equivalence queries in Line 3. The algorithm maintains the invariant $F \rightarrow G$. Hence, a counterexample $\mathbf{x}$ can only be a false-negative, i.e., $\mathbf{x} \not\models F$ but $\mathbf{x} \models G$. In principle, the counterexample $\mathbf{x}$ can be eliminated by updating F to $F \vee \mathbf{x}$ without executing the inner **for**-loop. However, in

Algorithm 2.3 DnfLearn: A DNF learning algorithm.

```
procedure DnfLearn($G(\overline{x})$), returns: A DNF representation $F(\overline{x})$ of $G(\overline{x})$
    $F :=$ false
    while EQ$(F, G)$ returns a counterexample $\mathbf{x}$ do
        $\mathbf{x}_g := \mathbf{x}$
        for each literal $l$ in $\mathbf{x}$ do
            if SUB$(\mathbf{x}_g \setminus \{l\}, G)$ then
                $\mathbf{x}_g := \mathbf{x}_g \setminus \{l\}$
            end if
        end for
        $F := F \vee \mathbf{x}_g$
    end while
    return $F$
end procedure
```

Algorithm 2.4 CnfLearn: A CNF learning algorithm.

```
procedure CnfLearn($G(\overline{x})$), returns: A CNF representation $F(\overline{x})$ of $G(\overline{x})$
    $F :=$ true
    while EQ$(F, G)$ returns a counterexample $\mathbf{x}$ do
        $\mathbf{x}_g := \mathbf{x}$
        for each literal $l$ in $\mathbf{x}$ do
            if SUB$(\mathbf{x}_g \setminus \{l\}, \neg G)$ then
                $\mathbf{x}_g := \mathbf{x}_g \setminus \{l\}$
            end if
        end for
        $F := F \wedge \neg\mathbf{x}_g$
    end while
    return $F$
end procedure
```

order to (potentially) reduce the number of iterations and also the size of F, the counterexamples are generalized: The inner loop drops literals from the cube $\mathbf{x}$ as long as the reduced cube $\mathbf{x}_g$ still implies G, i.e., represents only variable assignments that must be mapped to true in the end. Thus, the subsequent update $F := F \vee \mathbf{x}_g$ does not only eliminate the original counterexample $\mathbf{x}$, but may also eliminate many other counterexamples that have not been encountered yet. Note that this inner loop actually computes an unsatisfiable core $\mathbf{x}_g :=$ PropMinUnsatCore$(\mathbf{x}, \neg G)$. If no more counterexamples are left, the algorithm terminates and returns F, which is a disjunction of cubes, i.e., a DNF that is equivalent to G.

Learning a CNF. A CNF representation of a given formula $G(\overline{x})$ can be computed with $F = \neg$DnfLearn$(\neg G)$, i.e., by computing a DNF for $\neg G$ and negating the result. Alternatively, the procedure DnfLearn can easily be rewritten to compute CNFs directly. This is shown in Algorithm 2.4. The working principle remains the same, but F is initialized to true and refined with clauses that are computed from the false-positives returned by the equivalence queries.

More query learning algorithms can be found in the literature. For instance, an algorithm to learn formulas in form of a conjunction of DNFs can be defined using Bshouty's *monotone* theory [46]. In previous work [82], we show how various learning algorithms can be used effectively in circuit synthesis using BDDs. However, in this thesis we focus on SAT-based synthesis methods. SAT- and QBF solvers operate on CNF representations of a formula. Hence, our algorithms will mostly rely on the CNF learning approach. We therefore refrain from introducing more complicated learning methods here in detail, and refer the interested reader to the book by Crama and Hammer [68, Chapter 7].

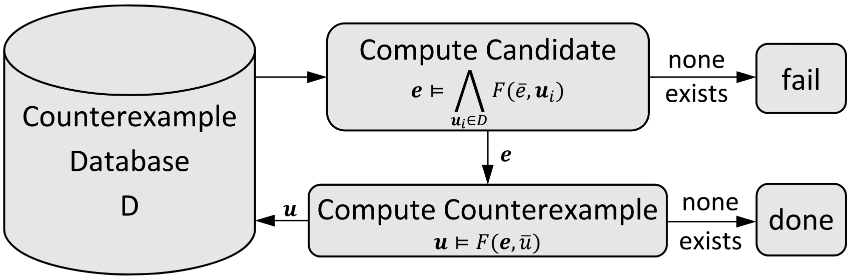

Counterexample
Database
D
Compute Candidate
$\mathbf{e} \models \bigwedge_{\mathbf{u}_i \in D} F(\overline{e}, \mathbf{u}_i)$
none
exists
fail
$\mathbf{e}$
$\mathbf{u}$
Compute Counterexample
$\mathbf{u} \models F(\mathbf{e}, \overline{u})$
none
exists
done

Figure 2.5: Working principle of Counterexample-Guided Inductive Synthesis (CEGIS). The goal is to compute a satisfying assignment $\mathbf{e}$ for $\exists \overline{e} : \forall \overline{u} : F(\overline{e}, \overline{u})$. This is done in a loop. There is a database D of counterexamples $\mathbf{u}_i$, which is initially empty. First, a candidate $\mathbf{e}$ that satisfies F for all $\mathbf{u}_i$ from D is computed. If none exists, the procedure fails. Otherwise, the candidate is checked. If no counterexample $\mathbf{u}$ is found, $\mathbf{e}$ is returned as solution. Otherwise, the counterexample $\mathbf{u}$ is added to D, and the loop is repeated.

2.7 Counterexample-Guided Inductive Synthesis (CEGIS)

The general principle of query learning, namely refining an initial "guess" of the solution iteratively based on counterexamples, has also been applied to other synthesis-related problems. One such example is Counterexample-Guided Inductive Synthesis (CEGIS) [197, 196], which was introduced in the context of program sketching as a method to compute satisfying assignments for quantified formulas of the form

$$\exists \overline{e} : \forall \overline{u} : F(\overline{e}, \overline{u}).$$

The goal is to compute concrete values $\mathbf{e}$ for the variables $\overline{e}$ such that $\forall \overline{u} : F(\mathbf{e}, \overline{u})$ holds. While the general principle is independent of the logic, we will assume that F is a quantifier-free first order logic formula that is to be interpreted modulo some background theory. Hence, $\overline{e}$ and $\overline{u}$ are vectors of domain variables, and we can use an SMT solver to reason about F (without the quantifiers). However, the CEGIS approach works analogously if F is a propositional formula over Boolean variables, in which case a SAT solver can be used. This thesis will use CEGIS in both these settings.

Working principle. Similar to query learning, a candidate $\mathbf{e}$ for a solution is iteratively refined based on counterexamples, which are concrete assignments to the variables $\overline{u}$ witnessing that $\forall \overline{u} : F(\mathbf{e}, \overline{u})$ does not yet hold. This refinement loop is illustrated in Figure 2.5. There is a database D of counterexamples $\mathbf{u}_i$, which is initially empty. The first step of the loop is to compute a candidate assignment

$$\mathbf{e} \models \bigwedge_{\mathbf{u}_i \in D} F(\overline{e}, \mathbf{u}_i)$$

that satisfies F for all counterexamples that have been encountered previously. This is a necessary but not a sufficient condition for $\forall \overline{u} : F(\mathbf{e}, \overline{u})$. Hence, if no such candidate $\mathbf{e}$ exists, this means that $\exists \overline{e} : \forall \overline{u} : F(\overline{e}, \overline{u})$ is unsatisfiable, so the algorithm aborts. If a candidate $\mathbf{e}$ was found, the next step is to check if $F(\mathbf{e}, \overline{u})$ holds*for all* $\overline{u}$ and not just for the concrete $\overline{u}$-values stored in D. This check is performed by searching for a counterexample $\mathbf{u} \models \neg F(\mathbf{e}, \overline{u})$ for which F does not (yet) hold with the given $\mathbf{e}$. If no such counterexample exists, then $\mathbf{e}$ must be a solution, and the algorithm terminates. Otherwise, the counterexample $\mathbf{u}$ is added to D and another iteration is performed. The candidate that is computed in the next iteration is already "better" in the sense that it satisfies F also for the counterexample from the previous iteration (and all iterations before). If the domain of the variables is finite, then the algorithm must terminate eventually. The reason is that every iteration excludes (at least) one candidate. Moreover, there is only a finite set of counterexamples to encounter.

Algorithm. The procedure CEGISSMT in Algorithm 2.5 implements the CEGIS algorithm using an SMT solver. Line 4 implements the candidate computation and Line 8 performs the candidate check as well as the counterexample computation in the straightforward way. Instead of storing a database

Algorithm 2.5 CegisSmt: The CEGIS algorithm implemented using an SMT solver.

```
 1: procedure CegisSmt($F(\overline{e}, \overline{u})$), returns: An assignment $\mathbf{e}$ for $\overline{e}$ such that $\forall \overline{u} : F(\mathbf{e}, \overline{u})$ or "fail"
 2:     $G(\overline{e}) := \mathsf{true}$
 3:     while true do
 4:         $(\mathsf{sat}, \mathbf{e}) := \textsc{SmtSatModel}\big(G(\overline{e})\big)$
 5:         if $\mathsf{sat} = \mathsf{false}$ then
 6:             return "fail"
 7:         end if
 8:         $(\mathsf{sat}, \mathbf{u}) := \textsc{SmtSatModel}\big(\neg F(\mathbf{e}, \overline{u})\big)$
 9:         if $\mathsf{sat} = \mathsf{false}$ then
10:             return $\mathbf{e}$
11:         end if
12:         $G(\overline{e}) := G(\overline{e}) \wedge F(\overline{e}, \mathbf{u})$
13:     end while
14: end procedure
```

of counterexamples, the algorithm directly refines the constraints for a candidate in Line 12. Note that constraints are only added to G, so the algorithm is well suited for incremental solving.

2.8 Software Program Analysis Techniques

In order to use automatic synthesis in a setting where parts of the program are already given, we need program analysis techniques to reason about the behavior of the existing program parts. In this thesis, we will use several kinds of analysis techniques. Symbolic and concolic execution will be used in an approach for automatic fault localization and correction in software. Furthermore, Hoare logic will be used in an alternative fault localization approach that reasons about individual functions in isolation instead of considering the entire program as a whole.

The following subsections will introduce these program analysis techniques in more detail. But before that, we will make a note on the undecidability of various program analysis problems.

2.8.1 Undecidability in Software Program Analysis

Many problems in software analysis are undecidable. This includes the question of whether a program satisfies a given specification. To overcome this issue, we will work with approximations.

Undecidable problems. The prime example of an undecidable problem is the *halting problem*, which asks to determine if a program terminates or continues to run forever for some given input. Alan M. Turing proved that this problem is undecidable in 1936 [209]. The halting problem can be reduced to many other interesting problems in software analysis in the sense that a decision procedure for some other problem P would give a decision procedure for the halting problem. Since such a decision procedure cannot exist, the reduction proves the undecidability of the problem P. One example is the question whether a given line in the source code is reachable for some given input: the program halts if and only if the last line is reached (assuming that the last line is the only exit point). Consequently, the decision problem to determine whether there exists some input for which a certain line in the source code is reached is undecidable as well (because the variant with a fixed input is only a special case). We refer the interested reader to Sipser [194] for more background on undecidability, the halting problem, and reductions.

Software verification is undecidable. In this thesis, we will mostly work with assertions in the code as a specification of the expected behavior of a program. An assertion is a statement `assert(c)`, which expresses that some condition c must be `true` at that point in the program. The question whether a given assertion can be violated is also undecidable. The reason is that the assertion can be rewritten to a branch `if(!c) exitErr();`, and the assertion can be violated if and only if the statement `exitErr()` is reachable. This means that in our setting, the problem of deciding whether a program satisfies its specification is undecidable. Our goal will be to repair incorrect software programs, but the problem of determining whether a repair eliminates the incorrectness is of course undecidable as well.

Workaround. As a general strategy out of this dilemma, we will apply program analysis in an incomplete and approximate way, e.g., by setting a bound on the number and length of execution paths to consider, and by approximating the semantics of individual statements. These approximations will also contribute towards achieving an acceptable level of scalability in practice.

2.8.2 Symbolic Execution

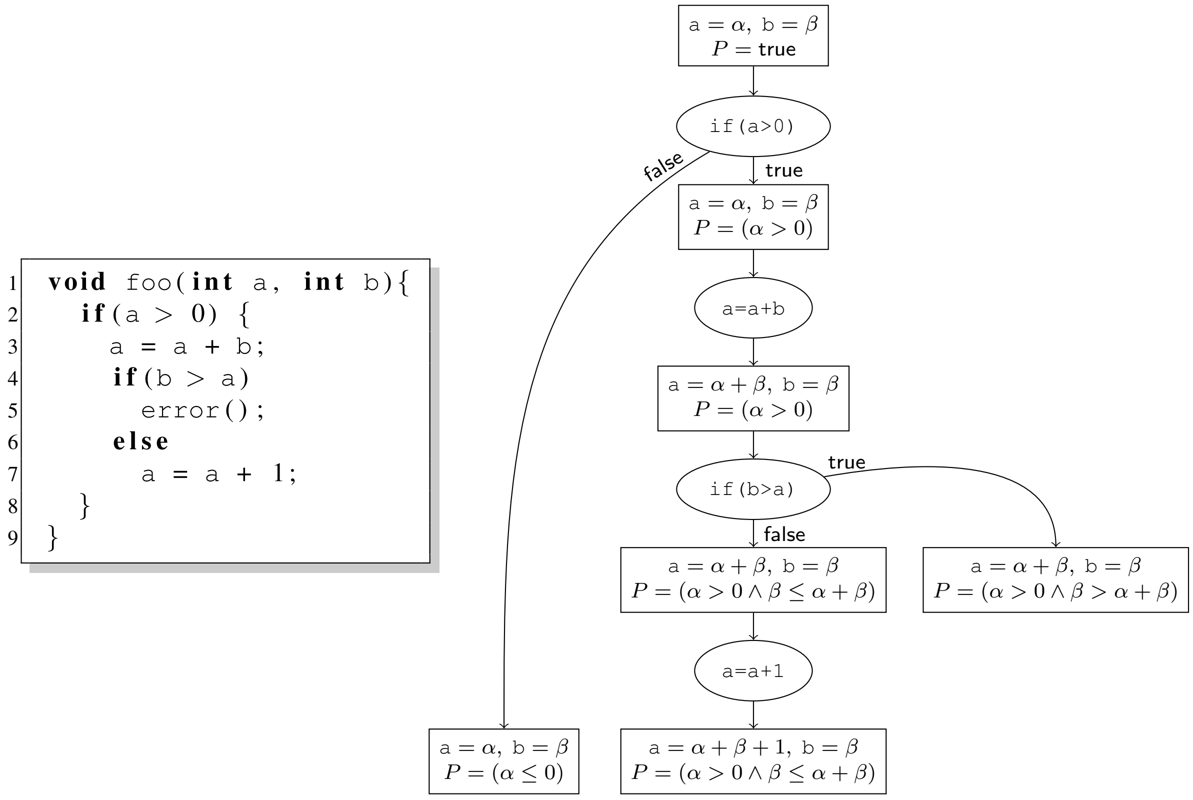
Symbolic execution [63, 136] is a technique to analyze the behavior of a software program under different inputs. It can be used to analyze under which circumstances some execution path through the program is activated, or to search for execution paths that reach certain points in the program code. In this thesis, we will use symbolic execution to compute an approximate condition that expresses when the program satisfies a given specification.

Working principle. The program is executed, but the execution is performed using symbols rather than concrete values for the inputs. The symbols are essentially just placeholders for any concrete value. In this sense, symbols are like variables. However, calling them *symbols* avoids confusions with the variables in a program. During the execution, all program variables have a symbolic value in form of an expression over the input symbols. Program statements update the symbolic value of the variables according to their semantics. Whenever a branching point in the program is reached, the symbolic execution forks. These forks unfold the execution into a *symbolic execution tree*. For each of the branches, a branching condition over the input symbols is computed. This branching condition expresses under which circumstances the respective branch is taken. Branching conditions are combined (by conjunction) into so-called *path conditions* along each execution path. Thus, a path condition expresses the circumstances under which a certain path through the program is taken. Whenever a path condition becomes unsatisfiable, this means that the corresponding execution path is infeasible. The execution path does not have to be explored further from such points, which prunes the symbolic execution tree.

Example 6. Figure 2.6 illustrates symbolic execution on an example. The left-hand side contains a program written in a C-like programming language. The right side shows the corresponding symbolic execution tree. Boxes denote the state of the symbolic execution between the program statements. The first line of each box contains the symbolic variable values. The second line gives the path condition P. Initially, the input variables `a` and `b` are set to the symbols α and β, which represent arbitrary values. At the `if` in Line 2, the execution forks. The branching conditions $\alpha > 0$ and $\alpha \leq 0$ are computed from the program expression `a>0` and the current symbolic value α of variable `a`. Because the path condition P is initially `true`, the path conditions of the successor nodes are equal to the branching conditions. In the `if`-branch, the next statement is `a=a+b`. It changes the symbolic value of variable `a` to $\alpha + \beta$. With the current symbolic values of `a` and `b`, the branching conditions for the `if` in Line 4 are $\beta > \alpha + \beta$ and $\beta \leq \alpha + \beta$. The path conditions of the successor nodes are the conjunctions with the path condition so far, i.e., with $\alpha > 0$. The path condition $\alpha > 0 \wedge \beta > \alpha + \beta$ for entering the second `if` is unsatisfiable[13]. This means that the `error()` function in Line 5 is unreachable. Therefore, the symbolic execution will not analyze this branch further. In the `else`-branch, the next statement to be executed is `a=a+1`. This statement changes the symbolic value of `a` again, and leaves `b` and P unchanged.

Discussion. The space of possible input values to a program can be huge or even infinite. In order to address this issue, symbolic execution does not exercise the program with concrete input values but keeps the inputs generic. However, at branching points in the program, symbolic execution forks and analyzes

[13] We ignore the possibility of variable overflows in this example.



```
void foo(int a, int b){
  if(a > 0) {
    a = a + b;
    if(b > a)
      error();
    else
      a = a + 1;
  }
}
```

Figure 2.6: A symbolic execution example. The left-hand side shows an example program in C syntax. The right-hand side shows the corresponding symbolic execution tree. Boxes illustrate the state between program statements. P denotes the path condition for reaching the node in the tree. The symbols α and β represent the arbitrary values of the input variables a and b, respectively.

each execution path separately. The number of execution paths to explore can grow large, especially if the program contains loops where the loop condition depends on inputs of the program. This problem is often referred to as *path explosion problem*. In order to make symbolic execution terminate within a reasonable amount of time, the maximum number and length of paths to explore can be bounded. Furthermore, heuristics can be applied in deciding which paths to explore first in order to achieve a high coverage quickly.

Tools. A prominent example of an open-source symbolic execution engine is KLEE [51], which is a redesign of the tool EXE [52]. KLEE operates on LLVM bytecode, for which a rich compiler infrastructure with front ends for many programming languages exists [152]. The symbolic reasoning is bit-level accurate and uses the SMT solver STP [95]. Various optimizations contribute to a compact representations of the symbolic states in order to save memory and to simplify queries for the SMT solver. Several heuristics are available for the order in which nodes of the symbolic execution tree are explored. For dangerous operations that could make the program crash (e.g., pointer dereferences or divisions) KLEE automatically checks if there exists values that are allowed by the current path condition and cause a failure. That is, the user (can but) does not have to write assertions or other means of specifications for KLEE to find bugs.

2.8.3 Concolic Execution

Pure symbolic execution can be quite resource demanding when analyzing large programs. One reason is that the number of branches in the symbolic execution tree can grow large, which results in a large memory consumption if many variables or memory locations need to be tracked symbolically. A more lightweight version of symbolic execution is *concolic execution* [99, 191], where the program is executed using concrete values and symbols at the same time. The artificial word "concolic" is a merge between "**conc**rete" and "symb**olic**" to express this hybrid nature.

Working principle. Concolic execution proceeds in several execution *runs*. Concrete input values for the first run are chosen arbitrarily. The program is executed with these concrete inputs, but in parallel to this execution, the symbolic values of the program variables are tracked, and a symbolic path condition is computed. The symbolic part of the execution works as explained in Section 2.8.2. However, the execution does not fork at the branching points, but only follows the branch that is determined by the concrete variable values. The result of one execution run is a path condition, which is a conjunction of the encountered branching conditions. This path condition is now analyzed in order to compute concrete input values that trigger a different execution path: one of the conjuncts is negated, all subsequent conjuncts are discarded, and a satisfying assignment is computed, e.g., using an SMT solver. The satisfying assignment defines the input values for the next run. In case of unsatisfiability, a different conjunct needs to be negated. This is repeated until all execution paths have been activated or some other termination criterion (e.g., a maximum number of iterations) is reached. Different strategies for negating conjuncts of the path conditions can be applied [50].

Example 7. For the program from Figure 2.6, we may choose the initial inputs $\texttt{a} = 0$ and $\texttt{b} = 0$. Hence, the first concolic execution run does not enter the `if` in Line 2 and yields the path condition $P_1(\alpha, \beta) = (\alpha \leq 0)$. This path condition contains only one conjunct that can be negated. Thus, a satisfying assignment for $\alpha > 0$ is computed next in order to activate a different execution path. Assume that the solver returns $\alpha = 1$ and $\beta = 0$, so the second run will have the input values $\texttt{a} = 1$ and $\texttt{b} = 0$. This second run enters the `if` in Line 2, but then follows the `else`-branch in Line 7. The resulting path condition is $P_2(\alpha, \beta) = (\alpha > 0 \wedge \beta \leq \alpha + \beta)$. The first part has already been negated. Negating the second part yields an unsatisfiable formula[14]. Hence, no more execution paths are feasible and the concolic execution terminates.

Advantages. The advantage of a potentially lower memory consumption of concolic execution compared to pure symbolic execution has already been mentioned. Another advantage is that the concrete variable values can be consulted if certain features of the programming language cannot be handled symbolically or if they shall deliberately be abstracted to improve the scalability. For instance, precise symbolic reasoning about the possible values of a pointer dereference operation may be difficult and computationally demanding for the underlying SMT solver. Taking the symbolic value associated with the concrete memory address that is stored in the pointer is an abstraction (because the pointer value may depend on the input) but can be a reasonable compromise between scalability and accuracy. A disadvantage of concolic execution in comparison to symbolic execution is that common execution path prefixes are analyzed multiple times.

Tools. Concolic execution is used in multiple tools and applications. Microsoft SAGE [100], where concolic execution is used to find potential security problems in parsers, has already been discussed in the introduction. Microsoft PEX [206] implements concolic execution for .NET programs and is integrated into the Visual Studio IDE. Other concolic execution tools include CUTE [191], DART [191], and CREST [50]. In our approach for fault correction, we will use an extension of CREST [50].

2.8.4 Hoare Logic

Hoare logic is a set of rules to prove correctness properties of a program. It has been introduced by C. A. R. Hoare [109] but was also influenced by earlier work of Floyd [88]. Similar concepts have also been proposed by Dijkstra [75].

Hoare triples and Hoare rules. The central concept in Hoare logic is a *Hoare triple* $\{P\}\ S\ \{Q\}$, where S is a program part and P and Q are formulas over program variables. The intuitive meaning of

[14] Assuming that we use Linear Integer Arithmetic (LIA), thereby ignoring the possibility for variables to overflow.

this Hoare Triple is that if P holds true before executing S, then Q will hold afterwards if S terminates. We will also refer to P as the *precondition* of S and to Q as the *postcondition* of S. The following paragraphs will introduce the most important rules for Hoare triples, which can be used to prove that a program satisfies some postcondition if some precondition holds and if it terminates. This is usually referred to as "partial correctness" because the fact that the program terminates still needs to be proven separately. We will write rules for Hoare triples as

$$\frac{\text{premises}}{\text{conclusions}} .$$

That is, if the premises above the line are satisfied, then the conclusions below the line follow. Axioms have an empty set of premises.

Assignment statements. Assignment statements are handled using the axiom

$$\frac{}{\{Q(e,\ldots)\}\ x = e;\ \{Q(x,\ldots)\}} , \tag{2.3}$$

where $x = e$; denotes the assignment of some expression e to variable x. The rule says that the formula Q holds after the assignment if (and only if) Q with x replaced by e holds before the assignment.

`if`-statements. For `if`-statements in the program, the following rule applies:

$$\frac{\{p \wedge P\}\ S_1\ \{Q\} \quad \{\neg p \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}(p)\ S_1\ \texttt{else}\ S_2\ \{Q\}} . \tag{2.4}$$

The premises say that the code in the `if`-branch and in the `else`-branch has the same postcondition Q. The preconditions $p \wedge P$ and $\neg p \wedge P$ only differ in the negation of the branching condition p. Then, as a consequence, P is the precondition of the entire `if`-statement and Q is the postcondition.

`while`-loops. In Hoare logic, `while`-loops require a loop invariant P, which is a formula that holds before and after every execution of the loop body. With such a loop invariant, the following rule can be applied:

$$\frac{\{p \wedge P\}\ S\ \{P\}}{\{P\}\ \texttt{while}(p)\ S\ \{\neg p \wedge P\}} . \tag{2.5}$$

The premise requires that P is indeed an invariant: If P holds before executing the loop body S, then P must also hold afterwards. By induction, this implies that P holds before and after every loop iteration. The loop condition p can also be assumed to hold before executing S because otherwise the execution would have left the loop already. As a conclusion of the rule, the loop invariant can be used as a precondition of the `while`-loop. The postcondition is the loop invariant with the negated loop condition p because the loop is only left if p is false.

Sequences of statements. The rule

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \tag{2.6}$$

for some statement S_1 followed by some other statement S_2 says that if the postcondition of S_1 matches the precondition of S_2, then the precondition of former is also a precondition of the composition, and the postcondition of the latter is also a corresponding postcondition of the composition. Here, S_1 and S_2 can be simple assignment statements, `if`-statements, `while`-loops, or sequences of statements themselves.

Consequence rule. The consequence rule

$$\frac{\{P_1\}\ S\ \{Q_1\} \quad P_2 \rightarrow P_1 \quad Q_1 \rightarrow Q_2}{\{P_2\}\ S\ \{Q_2\}} \tag{2.7}$$

expresses that preconditions can always be strengthened and postconditions can always be weakened. This rule can be useful as "glue" between other rules in order to make pre- and postconditions fit together.

Example 8. Consider the program `pow(a,b)`, which returns the value $x = a^b$ if $b \geq 0$.

```
 1  int pow(int a, int b) {
 2    $\{1 = a^0 \wedge 0 \leq b\} = \{b \geq 0\}$
 3    int x = 1;
 4    $\{x = a^0 \wedge 0 \leq b\}$
 5    int y = 0;
 6    $\{x = a^y \wedge y \leq b\}$
 7    while(y < b) {
 8      $\{x * a = a^{y+1} \wedge y + 1 \leq b\} = \{x = a^y \wedge y \leq b \wedge y < b\}$
 9      x = x * a;
10      $\{x = a^{y+1} \wedge y + 1 \leq b\}$
11      y = y + 1;
12      $\{x = a^y \wedge y \leq b\}$
13    }
14    $\{x = a^y \wedge y \leq b \wedge y \geq b\}$
15    $\{x = a^b\}$
16    return x;
17  }
```

The blue braces between the source code lines give intermediate pre- and postconditions. In the following, we will refer to individual pre- and postconditions and to source code statements with line numbers. For instance, $\{2\}\ 3\ \{4\}$ is short for the Hoare triple $\{0 \leq b\}\ x = 1;\ \{x = a^0 \wedge 0 \leq b\}$. Both $\{8\}\ 9\ \{10\}$ and $\{10\}\ 11\ \{12\}$ are valid Hoare triples by the assignment axiom from Equation 2.3. From these two Hoare triples, Equation 2.6 concludes $\{8\}\ 9; 11\ \{12\}$. With $P = (x = a^y \wedge y \leq b)$ and $p = (y < b)$ Equation 2.5 in turn concludes $\{6\}$ 7-13 $\{14\}$. Since $\{14\}$ implies $\{15\}$, Equation 2.7 concludes $\{6\}$ 7-13 $\{15\}$. The triples $\{2\}\ 3\ \{4\}$ and $\{4\}\ 5\ \{6\}$ again hold by Equation 2.3, and Equation 2.6 concludes $\{2\}\ 3; 5\ \{6\}$. In combination with $\{6\}$ 7-13 $\{15\}$, Equation 2.6 finally concludes $\{2\}$ 3-13 $\{15\}$. That is, if the precondition $b \geq 0$ holds before executing the code in `pow`, then $x = a^b$ will hold afterwards (given that `pow` terminates).

Verification. Hoare logic can be used to compute the weakest precondition $\mathsf{wp}(S, Q)$ under which a given postcondition (or assertion) Q of some code S holds. Most of this process is quite mechanic and can thus be automated. Only the application of the `while`-rule requires creativity in coming up with appropriate loop invariants. Tools such as the WP plug-in of the widely used software analysis tool suite Frama-C [69] automate this process of computing the weakest precondition but require the user to write loop invariants manually. In order to verify whether a given function f in the source code satisfies a contract in the form of a precondition P and a postcondition Q, an automatic (or interactive) theorem prover can then be used to check if the actual precondition P implies the computed weakest precondition, i.e., if $P \rightarrow \mathsf{wp}(f, Q)$ is valid. In Section 4.4, we will extend this verification concept with an approach for fault localization.

2.9 Model-Based Diagnosis

Model-Based Diagnosis (MBD) [72, 182] is a method for fault localization. It identifies potentially erroneous components of a system by explaining conflicts between a model of the system and an observation of some incorrect behavior. Based on the program analysis techniques presented in the previous section, we will use the principle of MBD to define a fault localization approach for software in Section 4.3. The resulting fault locations will in turn serve as basis for synthesizing repairs.

Terminology. We follow the fault localization terminology by Ammann and Offutt [6]. A *fault* is a "*static defect in the software*" or hardware program. For instance, a statement may read "`a=1;`" instead of "`a=0;`". We will also refer to a fault as a *bug* in the program. An *error* is an "*incorrect internal state that is the manifestation of some fault*". For instance, a certain variable may have a wrong value at some

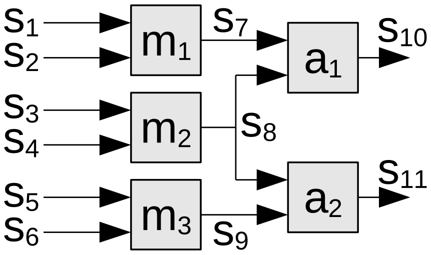


Figure 2.7: Example system for model-based diagnosis. The signals s_1 to s_6 are integer inputs, the signals s_{10} and s_{11} are integer outputs, the components m_1 to m_3 are multipliers, and the components a_1 and a_2 are adders.

point in the program. A *failure* is an "*external, incorrect behavior with respect to the requirements or other description of the expected behavior*". That is, a failure is a symptom of an error caused by a fault. For instance, the program produces wrong output or terminates unexpectedly because an assertion in the code is violated.

MBD Setup. For model-based diagnosis, we follow the notation by Reiter [182] in this thesis. A system description S is a formula in some logic and represents a model of a system. The system consists of a set of components CMP. A component $c \in \mathsf{CMP}$ can behave abnormally (denoted $\mathsf{AB}(c)$, where AB is a unary predicate) or normally (written $\neg\mathsf{AB}(c)$). Every component c is described with a formula of the form $\neg\mathsf{AB}(c) \rightarrow N_c$, where N_c is a formula that defines the normal behavior of component c. That is, abnormal components can behave arbitrarily. The system description S is composed of such component descriptions and constraints defining the interplay of components. Additionally, we are given an observation O of some erroneous behavior. This observation O is also given as a formula and contradicts S in the sense that, if all components behaved normally, it would be impossible to observe O. That is, the formula $\mathsf{S} \wedge \mathsf{O} \wedge \bigwedge_{c\in\mathsf{CMP}} \neg\mathsf{AB}(c)$ is unsatisfiable.

Example 9. Consider the system from Figure 2.7, which is also used as example in the work of Reiter [182]. It computes two integer outputs s_{10} and s_{11} based on the integer inputs s_1 to s_6. The system has 5 components $\mathsf{CMP} = \{m_1, m_2, m_3, a_1, a_2\}$, where m_1 to m_3 are multipliers and a_1 and a_2 are adders. The system description can be defined as

$$\begin{aligned}\mathsf{S} =& \big(\neg\mathsf{AB}(m_1) \rightarrow (s_7 = s_1 \cdot s_2)\big) \wedge \big(\neg\mathsf{AB}(m_2) \rightarrow (s_8 = s_3 \cdot s_4)\big) \wedge \big(\neg\mathsf{AB}(m_3) \rightarrow (s_9 = s_5 \cdot s_6)\big) \wedge \\ & \big(\neg\mathsf{AB}(a_1) \rightarrow (s_{10} = s_7 + s_8)\big) \wedge \big(\neg\mathsf{AB}(a_2) \rightarrow (s_{11} = s_8 + s_9)\big).\end{aligned}$$

An observation is given as $\mathsf{O} = (s_1 = s_3 = s_5 = 3) \wedge (s_2 = s_4 = s_6 = 2) \wedge (s_{10} = 10) \wedge s_{11} = 12)$. The formula $\mathsf{S} \wedge \mathsf{O} \wedge \neg\mathsf{AB}(m_1) \wedge \neg\mathsf{AB}(m_2) \wedge \neg\mathsf{AB}(m_3) \wedge \neg\mathsf{AB}(a_1) \wedge \neg\mathsf{AB}(a_1)$ is unsatisfiable because if all components behave normally, the computed output is $s_{10} = s_{11} = 12$, but $s_{10} = 10$ was observed.

Diagnoses. The goal of MBD is to compute diagnoses, which are sets of components that may be responsible for the observation O. Formally, a *diagnosis* is a set $\Delta \subseteq \mathsf{CMP}$ such that $\mathsf{S} \wedge \mathsf{O} \wedge \bigwedge_{c\in\mathsf{CMP}\setminus\Delta} \neg\mathsf{AB}(c)$ is satisfiable. That is, assuming that the components in Δ behave abnormally renders the observation O possible. A diagnosis Δ is *minimal* if no subset $\Delta' \subset \Delta$ is a diagnosis. If Δ is a diagnosis, then clearly every superset $\Delta' \supseteq \Delta$ is a diagnosis as well. Hence, we are only interested in computing minimal diagnoses. A diagnosis Δ with $|\Delta| = 1$ is called a *Single-Fault Diagnosis (SFD).*

Conflicts. Diagnoses can be computed via conflicts. A *conflict* is a set $C \subseteq \mathsf{CMP}$ of components such that $\mathsf{S} \wedge \mathsf{O} \wedge \bigwedge_{c\in C} \neg\mathsf{AB}(c)$ is unsatisfiable. Thus, a conflict is a set of components that cannot all behave normally. If they would, the observation would have been impossible. A conflict C is called *minimal* if no subset $C' \subset C$ is a conflict.

Example 10. For the system description S and observation O from Example 9, there exist two minimal conflicts, namely $C_1 = \{m_1, m_2, a_1\}$ and $C_2 = \{m_1, m_3, a_1, a_2\}$. The set C_1 is a conflict because if m_1, m_2 and a_1 behave normally, then $s_7 = s_8 = 6$ and $s_{10} = 12$, which contradicts $s_{10} = 10$ in the observation. The set C_2 is a conflict because if m_2 behaves abnormally then s_8 can be arbitrary, but if all other components behave normally we have $s_{10} = s_8 + 6$ and $s_{11} = s_8 + 6$, i.e., $s_{10} = s_{11}$. This contradicts $s_{10} = 10$ and $s_{11} = 12$ in our observation. There are 4 minimal diagnoses. The set $\{a_1\}$ is a diagnosis because a_1 could have computed $6+6 = 10$ if it behaved abnormally, which would explain the observation. If component m_1 behaved abnormally and computed $2 \cdot 3 = 4$ then this would explain the observation as well. Hence, $\{m_1\}$ is a second diagnosis. The set $\{m_2, m_3\}$ is another diagnosis because these components could have computed $2 \cdot 3 = 4$ and $2 \cdot 3 = 8$, respectively. Finally, $\{m_2, a_2\}$ is a diagnosis because these components could have computed $2 \cdot 3 = 4$ and $4 + 6 = 12$.

Hitting sets. The relation between conflicts and diagnoses can be expressed using hitting sets. A *hitting set* for a collection $\mathcal{K}$ of sets is a set H such that $\forall K \in \mathcal{K} : H \cap K \neq \emptyset$ holds. That is, the hitting set H must have at least one element in common with *every* set K in $\mathcal{K}$. A hitting set H is called *minimal* if no subset $H' \subset H$ is a hitting set for $\mathcal{K}$.

Computing diagnoses. A set $\Delta \subseteq \mathsf{CMP}$ is a minimal diagnosis iff Δ is a minimal hitting set for the collection of all conflicts [182]. The intuitive reason is that a diagnosis must explain all conflicts, so it must share at least one element with every conflict. Since every conflict is a superset of at least one minimal conflict, it is sufficient to compute minimal hitting sets for the collection of all minimal conflicts. Reiter [182] presents a hitting set computation algorithm which computes conflicts on the fly and produces diagnoses in the order of increasing cardinality. Thus, if the algorithm is aborted before all diagnoses have been computed, only diagnoses with higher cardinality (which are generally considered as less likely) are missed. We refrain from repeating the hitting set computation algorithm by Reiter in this thesis, but we will use it as a black box building block for our fault localization approach in Section 4.3.

3 SAT-Based Hardware Controller Synthesis

Parts of this chapter are based on my previous publications [82, 39, 31] as well as on results from the Bachelor's Thesis of Patrick Klampfl [137], which I co-supervised. References to these sources are not always made explicit.

In this section, we present our SAT-based hardware controller synthesis approach for safety specifications. Similar to the standard textbook method presented in Section 2.5.3, our approach consists of two steps. First, a winning strategy is computed. Second, a circuit implementing the strategy is generated. Various SAT-based methods to solve the first step will be presented in Section 3.1. SAT-based methods to realize the second step will be presented in Section 3.2. Finally, Section 3.3 will present experimental results.

3.1 From Safety Specifications to Strategies

As discussed in Section 2.5.3, a strategy S for realizing a given safety specification $\mathcal{S} = \big(\overline{x}, \overline{i}, \overline{c}, I(\overline{x}), T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'), P(\overline{x})\big)$ can be constructed by computing the winning region $W(\overline{x})$ in the game defined by $\mathcal{S}$. Recall that the winning region is the set of all states from which the system player can enforce that only safe states are visited. Once the winning region is available, the corresponding strategy can be defined by

$$S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') = T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(W(\overline{x}) \rightarrow W(\overline{x}')\big).$$

However, a winning strategy can also be computed by different means. One option is to use a winning area, defined as follows.

Definition 11 (Winning Area)**.** *A* winning area *for a safety specification $\mathcal{S} = \big(\overline{x}, \overline{i}, \overline{c}, I(\overline{x}), T(\overline{x}, \overline{i}, \overline{c}, \overline{x}'), P(\overline{x})\big)$ is a state set F, represented symbolically as a formula $F(\overline{x})$, with the following three properties:*

- *Every initial state is contained in F, i.e., $I(\overline{x}) \rightarrow F(\overline{x})$.*
- *F contains only safe states, i.e., $F(\overline{x}) \rightarrow P(\overline{x})$.*
- *The system player can enforce that the play stays in F, i.e., $F(\overline{x}) \rightarrow \mathsf{Force}_1^s\big(F(\overline{x})\big)$.*

These three properties are sufficient to ensure that $T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(F(\overline{x}) \rightarrow F(\overline{x}')\big)$ is a winning strategy. The reason is the same as for the winning region (see Section 2.5.3.1): the control signals can always be chosen in such a way that the next state is in F again, and F contains only safe states. In fact, the winning region is just a special winning area, namely the largest one. The following sections will present different methods for computing the winning region or a winning area.

3.1.1 QBF-Based Learning

The SafeWin procedure presented in Algorithm 2.1 can be implemented with BDDs using their capability of quantifier elimination in a rather straightforward manner. However, a realization with plain SAT solvers is not easily possible because the preimage operation Force_1^s in Line 4 contains a universal quantification. Therefore, a natural option is to use a QBF solver, which can handle universal quantifications without expanding the formula.

3.1.1.1 A Straightforward QBF Realization of SafeWin

A direct realization of SafeWin with QBF solving was presented by Staber and Bloem [201]. We briefly review this existing method and its drawbacks before presenting our learning-based algorithms. For this discussion, we will refer to the different values of the variable F in Algorithm 2.1 with indices.

Algorithm 3.1 QBFWIN: Basic QBF-based CNF learning algorithm for the winning region.

```
1:  procedure QBFWIN($((\overline{x}, \overline{i}, \overline{c}, I, T, P))$), returns: The winning region $W(\overline{x})$ in CNF or false
2:      if PROPSAT($I(\overline{x}) \wedge \neg P(\overline{x})$) then
3:          return false
4:      end if
5:      $F(\overline{x}) := P(\overline{x})$
6:      // check if $F \rightarrow \mathsf{Force}_1^s(F)$ is valid:
7:      while sat in $(\mathsf{sat}, \mathbf{x}) :=$ QBFSATMODEL($\exists \overline{x}, \overline{i} : \forall \overline{c} : \exists \overline{x}' : F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')$) do
8:          // generalize the counterexample:
9:          $\mathbf{x}_g := \mathbf{x}$
10:         for each literal $l$ in $\mathbf{x}$ do
11:             $\mathbf{x}_t := \mathbf{x}_g \setminus \{l\}$
12:             if ¬QBFSAT($\exists \overline{x} : \forall \overline{i} : \exists \overline{c}, \overline{x}' : \mathbf{x}_t \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$) then
13:                 $\mathbf{x}_g := \mathbf{x}_t$
14:             end if
15:         end for
16:         // check for unrealizability:
17:         if PROPSAT($\mathbf{x}_g \wedge I(\overline{x})$) then
18:             return false
19:         end if
20:         // refine $F$:
21:         $F(\overline{x}) := F(\overline{x}) \wedge \neg \mathbf{x}_g$
22:     end while
23:     return $F(\overline{x})$
24: end procedure
```

That is, $F_0 = P$ denotes the initial value of F and $F_j = F_{j-1} \wedge \mathsf{Force}_1^s(F_{j-1})$ is the value after the j^{th} iteration. The termination check in Line 3 is performed by checking two subsequent values F_j and F_{j-1} for equivalence. Since $F_j \rightarrow F_{j-1}$, i.e., the set F of states can only get smaller from iteration to iteration, it is sufficient to check if $F_{j-1} \rightarrow F_j$. Thus, the first check of "F changes" can be realized with the QBF query

$$\neg\text{QBFSAT}\Big(\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : P(\overline{x}) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge P(\overline{x}')\big)\Big).$$

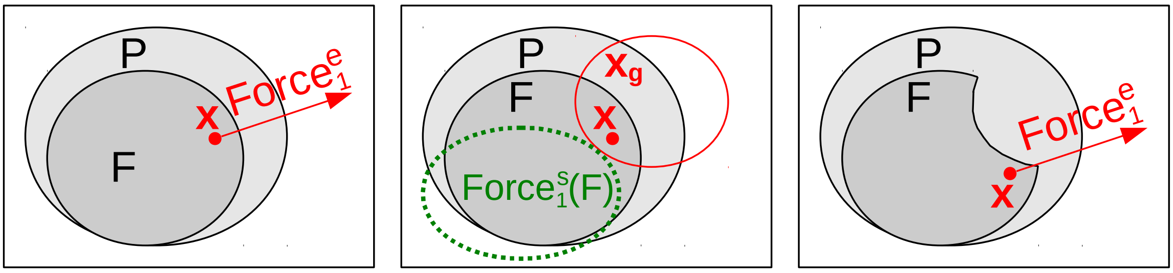
The second check if F changes translates to

$$\neg\text{QBFSAT}\Big(\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : \forall \overline{i}' : \exists \overline{c}', \overline{x}'' : \big(P(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge P(\overline{x}')\big) \rightarrow \big(T(\overline{x}', \overline{i}', \overline{c}', \overline{x}'') \wedge P(\overline{x}'')\big)\Big).$$

In general, the check if F changed in iteration j requires solving a QBF with $2 \cdot j - 1$ quantifier alternations and j copies of the transition relation T. The checks if $I \rightarrow F_j$ in Line 5 of Algorithm 2.1 work in a similar way, also requiring $2 \cdot j - 1$ quantifier alternations and j copies of the transition relation. We consider this steep increase in formula size and complexity as suboptimal. In the following, we will therefore present algorithms that require only one copy of the transition relation and a constant number of quantifier alternations in the queries to the QBF solver.

3.1.1.2 A QBF-Based CNF Learning Algorithm

Algorithm 3.1 shows the procedure QBFWIN, which computes a CNF representation of the winning region $W(\overline{x})$ using CNF learning with a QBF solver. Since QBFWIN will also be the basis for our algorithms that use plain SAT solving, we discuss it here in detail. Just like SAFEWIN in Algorithm 2.1, QBFWIN takes a specification as input. It returns either the winning region $W(\overline{x})$ or false in case of unrealizability. The basic structure is that of the CNF learning procedure CNFLEARN in Algorithm 2.4.



(a) Counterexample computation. **(b)** Generalization. **(c)** Update of F.

Figure 3.1: Working principle of QBFWIN. Subfigure (a) illustrates the computation of a counterexample $\mathbf{x} \models F \wedge \mathsf{Force}_1^e(\neg F)$. Subfigure (b) shows the generalization of $\mathbf{x}$ into a larger region $\mathbf{x}_g$ that does not intersect with $\mathsf{Force}_1^s(F)$ in F. Subfigure (c) illustrates the refinement of F by removing $\mathbf{x}_g$, as well as the next counterexample computation.

However, in Line 5, F is initialized to P instead of true because the winning region can only be a subset of the safe states P. Differences in counterexample computation and generalization are discussed in the following.

Counterexample computation. The equivalence query in Line 3 of the original CNF learning procedure CNFLEARN asks if the current approximation of the solution is correct. The corresponding line (Line 7) in QBFWIN now checks if $F \rightarrow \mathsf{Force}_1^s(F)$ is valid, i.e., if another visit of F can be enforced by the system from any state of F. The QBF query in Line 7 of QBFWIN actually asks the opposite question, namely if there exists a state $\mathbf{x}$ in F from which the environment can enforce leaving F, i.e., if $F \wedge \mathsf{Force}_1^e(\neg F)$ is satisfiable. This is the case if there exists some state $\mathbf{x}$ in F and some input $\mathbf{i}$ such that for all control values $\mathbf{c}$ the next state will be in $\neg F$. If such a state $\mathbf{x}$ exists, QBFSATMODEL will return it as a counterexample witnessing that F is not equal to the final winning region W. More specifically, this counterexample state $\mathbf{x}$ cannot be part of W, and thus needs to be removed from F. This removal is performed in Line 21. However, in order to reduce the number of iterations, the counterexample is generalized before it is excluded. This is explained in the next paragraph. If, on the other hand, QBFSATMODEL sets sat to false in Line 7, then this means that the implication $F \rightarrow \mathsf{Force}_1^s(F)$ holds. In this case, the algorithm terminates, returning F as the winning region.

Counterexample generalization. Just like in CNFLEARN, counterexample generalization is done by eliminating literals of $\mathbf{x}$ in the inner loop of the algorithm. In CNFLEARN (see Algorithm 2.4), the final cube $\mathbf{x}_g \subseteq \mathbf{x}$ must not intersect with G in order not to shrink F beyond G. Similarly, in QBFWIN, $\mathbf{x}_g \wedge F$ must not intersect with $\mathsf{Force}_1^s(F)$ in order not to remove any states from the winning region where the system could enforce that the play stays in the winning region. The reason is that the subsequent update $F := F \wedge \neg\mathbf{x}_g$ in Line 21 removes exactly the states $\mathbf{x}_g \wedge F$. The QBF query in Line 12 is satisfiable if $\mathbf{x}_t \wedge F$ contains any states of $\mathsf{Force}_1^s(F)$, and thus prevents unjust state removals. Also note that the inner loop essentially computes an unsatisfiable core of $\mathbf{x}$ with respect to $F \wedge \mathsf{Force}_1^s(F)$.

Detecting unrealizability. Detecting unrealizability is simple. The specification is unrealizable if and only if some initial state is outside of the winning region, i.e., if $I \not\rightarrow W$. The reason is that no system implementation can prevent the environment from visiting an unsafe state from an initial state that is not winning. QBFWIN returns false as soon as $I \not\rightarrow F$. Since $F = W$ eventually, this ensures that false is returned if $I \not\rightarrow W$. Line 2 checks if $I \not\rightarrow F$ would hold initially. In every iteration, Line 17 then checks if the states $\mathbf{x}_g$ that are going to be removed from F contain an initial state. This is potentially more efficient than than checking $I \not\rightarrow F$ again.

Illustration. Figure 3.1 illustrates the working principle of QBFWIN graphically. A box represents the set of all states. F is always a subset of P. In Figure 3.1a, a counterexample $\mathbf{x} \models F \wedge \mathsf{Force}_1^e(\neg F)$ is computed. It represents a state from which the environment can enforce that F is left. Next, the counterexample $\mathbf{x}$ is generalized into a larger region $\mathbf{x}_g$ by eliminating literals, as illustrated in Figure 3.1b. Every literal that can be eliminating from $\mathbf{x}$ doubles the size of the state region that is represented by

$\mathbf{x}_g$. Literals are dropped as long as $\mathbf{x}_g \wedge F$ does not intersect with $\mathsf{Force}_1^s(F)$. Finally, as illustrated in Figure 3.1c, the generalized counterexample $\mathbf{x}_g$ is removed from F and the next counterexample is computed. This is repeated until no more counterexamples exist, or one of the initial states is removed.

The following theorem summarizes these explanations into a formal correctness argument.

Theorem 12. *The* QBFWIN *procedure in Algorithm 3.1 returns the winning region $W(\overline{x})$ of a given safety specification $\mathcal{S}$, or* false *if the specification is unrealizable.*

Proof. QBFWIN enforces the invariants $F \rightarrow P$ (through Lines 5 and 21) and $I \rightarrow F$ (through Lines 2 and 17). The loop terminates normally if $F \rightarrow \mathsf{Force}_1^s(F)$. Hence, upon normal termination, F is certainly a winning area according to Definition 11. F is also the largest possible winning area, and thereby the winning region, because QBFWIN also enforces the invariant $W \rightarrow F$. This invariant can be proven by induction: Initially $F = P$, so $W \rightarrow F$ holds because $W \rightarrow P$. Under the hypothesis that $W \rightarrow F$ holds before an update of F in Line 21, it will also hold after the update because Line 21 only removes states $\mathbf{x}_g \wedge F$ for which $\mathbf{x}_g \wedge F \rightarrow \mathsf{Force}_1^e(\neg F)$ holds. Given that $W \rightarrow F$, we have that $\neg F \rightarrow \neg W$. This means that $\mathbf{x}_g \wedge F \rightarrow \mathsf{Force}_1^e(\neg W)$, so only states that cannot be part of W are removed. QBFWIN will always terminate because in every iteration, at least one state is removed from F, and when F reaches false (or earlier) the loop necessarily terminates. What remains to be shown is that QBFWIN aborts in Line 2 or 17 iff $\mathcal{S}$ is unrealizable, i.e., iff $I \not\rightarrow W$. (Direction ⇒:) Since $W \rightarrow F$, and Line 2 or 17 abort iff (F is about to be updated in such a way that) $I \not\rightarrow F$, it follows that QBFWIN can only abort if $I \not\rightarrow W$. (Direction ⇐:) Since $F = W$ eventually, Line 2 or 17 will definitely abort eventually if $I \not\rightarrow W$. □

Discussion. In contrast to the approach sketched in Section 3.1.1.1, all QBF queries in QBFWIN contain only one copy of the transition relation and only two quantifier alternations. This potentially increases the scalability with respect to the size of the specifications. The disadvantage is that the number of calls to the QBF solver can be significantly higher.

3.1.1.3 Variants and Improvements

In this section, we now discuss a few variants and optimizations of QBFWIN as presented in Algorithm 3.1.

Better generalization. At any point in the inner loop of QBFWIN, $\mathbf{x}_g$ represents states that will definitely be removed from F. This information can be exploited already during the generalization loop by modifying the QBF query in Line 12 to

$$\neg\mathrm{QBFSAT}\big(\exists \overline{x} : \forall \overline{i} : \exists \overline{c}, \overline{x}' : \mathbf{x}_t \wedge F(\overline{x}) \wedge \neg \mathbf{x}_g \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}') \wedge \neg \mathbf{x}_g'\big).$$

This way, the generalization loop behaves as if F would have been refined to $F(\overline{x}) \wedge \neg \mathbf{x}_g$ already (with the current version of $\mathbf{x}_g$). The QBF query becomes stricter, which can have the effect that more literals can be eliminated. This can reduce the total number of counterexamples that have to be resolved. In the illustration of Figure 3.1b, this optimization shrinks $\mathsf{Force}_1^s(F)$ to $\mathsf{Force}_1^s(F \wedge \mathbf{x}_g)$, which allows $\mathbf{x}_g$ to grow even larger. Since this optimization does not increase the number or complexity of the QBF queries, we always apply it.

Generalization until fixpoint. With the generalization optimization from the previous paragraph, the generalization check becomes non-monotonic in the sense that, even if a literal could not be eliminated initially, it may be eliminable after eliminating other literals. Hence, it can be beneficial to repeat the generalization loop until a fixpoint is reached. However, in our experiments, this did not result in noticeable performance improvements on the average over our benchmarks, so this is not done by default.

Computing all generalizations. In our experiments we observed that counterexample computation often takes much more time than counterexample generalization. Moreover, depending on the order in which the literals $l \in \mathbf{x}$ are processed in Line 10 of QBFWIN, we can get different generalizations $\mathbf{x}_g$.

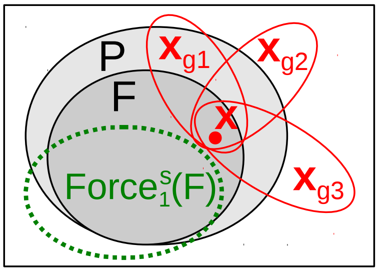


Figure 3.2: Computing all generalizations of a counterexample in QBFWIN. The counterexample is drawn as a red dot. The generalizations $\mathbf{x}_{g1}$, $\mathbf{x}_{g2}$ and $\mathbf{x}_{g3}$ are drawn as red ellipses. Removing all generalization prunes F more than removing just one.

Motivated by these observations, we propose a variant that computes *all* minimal generalizations for each counterexample. A naive solution would just run the generalization loop of Line 10 repeatedly using all $|\mathbf{x}|!$ different orders of the literals in $\mathbf{x}$. However, since many orderings can result in the same generalization $\mathbf{x}_g$, this is potentially inefficient. Instead, we thus apply an adaption of the hitting set tree algorithm presented by Reiter [182]. For the sake of readability, we refrain from presenting this algorithm in detail. The high-level intuition is visualized in Figure 3.2. All generalizations $\mathbf{x}_{g1}$, $\mathbf{x}_{g2}$ and $\mathbf{x}_{g3}$ will contain the original counterexample $\mathbf{x}$, and none of them may intersect with $\mathsf{Force}_1^s(F)$ inside of F. Although there may be a significant overlap between the generalizations, removing all of them prunes F more than removing just one of them. In our experiments, we observed that the number of different counterexample generalizations is usually low. Not infrequently, there is only exactly one minimal generalization. Of course, computing all generalizations costs additional computation time. In our experiments, it gives a solid speedup for some benchmarks, but slows down the computation for others. Hence, we do not apply this optimization by default.

Instead of computing all generalizations, one could also compute and apply at most k different generalizations for some value of k. Another option is to compute all generalizations but refine F only with the k shortest ones. However, in preliminary experiments, these variants did not result in significant performance increases either.

3.1.1.4 Efficient Implementation

In this section, we give a few remarks on implementing QBFWIN efficiently.

CNF encoding. The transition relation T, the characterization of the safe states P and the formula for the initial states I are transformed into CNF initially. Furthermore, a CNF representation of $\neg F$ needs to be computed in each iteration. All these transformations can be done using the method of Plaisted and Greenbaum [174]. This may introduce additional auxiliary variables, which are quantified existentially on the innermost level of the QBF queries. Once T, P, I and $\neg F$ are available in CNF, the matrices of the QBF queries in Algorithm 3.1 can be constructed by building the union of the respective clause sets, because the individual formula parts are all connected by conjunctions.

CNF compression. After some iterations, the CNF formula F in QBFWIN can contain redundant clauses and literals. First, a clause discovered in some later iteration can be a proper subset of some earlier discovered clause. This can be checked syntactically at low costs. Thus, whenever a clauses is added to F, we always remove all of its supersets. Second, a set of clauses may together imply clauses that have been added earlier. The implied clauses can be eliminated without changing F semantically. Third, it may be possible to drop literals from clauses of F in an equivalence-preserving manner. The procedure COMPRESSCNF in Algorithm 3.2 performs these simplifications and is explained in the next paragraph. We call this procedure to simplify F after every modification of F, but with literal dropping disabled (we will later use COMPRESSCNF with literal dropping enabled in other contexts). COMPRESSCNF is very fast compared to the QBF solver calls in QBFWIN. Furthermore, a smaller CNF representation of F is particularly important for computing a compact representation of $\neg F$ using the method of Plaisted

Algorithm 3.2 CompressCnf: Removing redundant literals and clauses from a CNF.

```
procedure CompressCnf($F(\overline{x})$), returns: An equivalent but potentially smaller CNF $G(\overline{x})$
    if dropping literals enabled then
        $G :=$ true
        for each clause $c$ in $F$ do
            $G := G \wedge \neg$PropMinUnsatCore$(\neg c, F)$
        end for
        $F := G$
    end if
    $G :=$ true
    for each clause $c$ in $F$ with increasing size do
        if PropSat$(G \wedge \neg c)$ then
            $G := G \wedge c$
        end if
    end for
    return $G$
end procedure
```

and Greenbaum [174]. Ultimately, the more compact CNF representations reduce the QBF solving time quite significantly.

An algorithm for CNF compression. Algorithm 3.2 uses a SAT solver to remove redundant literals and clauses from a CNF formula $F(\overline{x})$. The first loop (if enabled) drops literals from each clause c as long as the reduced clause $c_2 \subseteq c$ is still implied by F. This ensures that the reduced formula G is implied by F. Dropping literals can only make the formula stronger, i.e., F is necessarily implied by G. Hence, G and F are equivalent. Note that $F \rightarrow c_2$ iff $F \wedge \neg c_2$ is unsatisfiable. Hence, dropping the literals can be realized by computing a (minimal) unsatisfiable core of the cube $\neg c_2$ with respect to F. Since F does not change in this loop, all unsatisfiable cores can be computed with incremental SAT solving.
The second loop removes redundant clauses. Non-redundant clauses are copied into G. A clause c is redundant if it is implied by G already, i.e., if $G \wedge \neg c$ is unsatisfiable. Clauses are processed in the order of increasing size because smaller clauses have a higher tendency to imply larger clauses than the other way around. This second loop can also be accomplished with incremental solving, since clauses are only added to G. Dropping literals before eliminating clauses potentially yields better results than performing the operations in the reverse order. The reason is that the shorter clauses produced in the first loop have a higher potential for implying other clauses in the second loop. Since none of the SAT solver calls involves the transition relation, Algorithm 3.2 is usually very fast. It will not only be used in QbfWin, but also in other contexts.

QBF preprocessing. Our work with Martina Seidl on extending the popular QBF preprocessor Bloqqer to preserve satisfying assignments [189] enables QBF preprocessing not only in QbfSat, but also in QbfSatModel queries. We thus apply QBF preprocessing to every single QBF query (separately). The experimental results in Section 3.3 will show that this is crucial for the performance. In a sense, running CompressCnf to simplify F, as explained in the previous paragraphs, can also be seen as QBF preprocessing, but using knowledge about the structure of the final QBF. Bloqqer [26] implements way more simplification techniques, from heuristics for universal expansion to variable elimination, and is thus clearly not subsumed by running CompressCnf. On the other hand, our experiments indicate that running CompressCnf in addition to Bloqqer is beneficial as well. A possible reason is that we compress F *before* computing its negation. This has advantages over applying simplifications on the final QBF, where the structure is already lost.

Incremental QBF solving. We experimented with incremental QBF solving using DepQBF [156]. We use two incremental solver instances, one for the queries in Line 7 and one for Line 12 of QbfWin.

The queries in Line 12 are well suited for incremental solving because clauses are only added to F. The conjunction with $\mathbf{x}_t$ can be achieved with assumption literals, which are temporarily asserted. In fact, we first let DepQBF compute an unsatisfiable core of $\mathbf{x}$ and minimize this core then further using a loop that attempts to eliminate more literals.
The check in Line 7 of QBFWIN is more difficult because it also contains the negation of the F, i.e., cannot be realized incrementally just by adding additional clauses. We implemented three variants to handle $\neg F(\overline{x}')$ incrementally. Since neither of these three variants performs particularly well in our experiments (see Section 3.3), we only sketch them briefly. The first variant uses the push/pop interface of DepQBF to replace the parts in the CNF encoding of $\neg F(\overline{x}')$ that change from iteration to iteration. The second variant updates $\neg F(\overline{x}')$ only lazily, namely when the check in Line 7 becomes unsatisfiable.[1] In this event, a new incremental session of the solver is started with the latest version of $\neg F(\overline{x}')$. The third variant uses a pool of variables to encode negated clauses in CNF. If a variable of this pool is not yet used, it is set to false using assumption literals. Thereby, the variable essentially represents the negation of a tautological clause. As clauses are added to F, the variables of the pool are equipped with constraints that make them represent the negation of the added clauses. If there are no more unused variables in the pool, a new incremental session with a fresh pool of variables is started. As already mentioned, neither of these three variants performs particularly well in our experiments. One reason is that incremental QBF solving cannot be combined with preprocessing at the moment. However, this may change in the future, which could make these approaches interesting again.

3.1.2 Learning Based on SAT Solving

In this section, we will present a learning algorithm that computes the winning region of a safety specification $\mathcal{S} = (\overline{x}, \overline{i}, \overline{c}, I, T, P)$ using a plain SAT solver. In order to simplify the presentation, this will be done in two steps: Section 3.1.2.1 presents a basic algorithm. Section 3.1.2.2 will then discuss a more efficient variant with better support for incremental SAT solving.

3.1.2.1 Basic Algorithm

A basic solution is shown in Algorithm 3.3. The working principle is the same as for the procedure QBFWIN from Algorithm 3.1: starting with the initial over-approximation $F = P$ of the winning region W, counterexample-states $\mathbf{x} \models F \wedge \mathsf{Force}_1^e(\neg F)$ witnessing that $F \neq W$ are computed, generalized into a larger region $\mathbf{x}_g$ of states that cannot be part of the final winning region W, and finally removed from F. Detecting unrealizability by checking if $I \not\rightarrow F$ is also done in exactly the same way as in QBFWIN. Only the counterexample computation and generalization is different, and will be discussed in the following paragraphs.

Counterexample computation. We need to find a state $\mathbf{x}$ from which the environment can enforce that F is left. That is, from state $\mathbf{x} \models F$, there must exist some input $\mathbf{i}$ such that for all control values $\mathbf{c}$, the next state will satisfy $\neg F$. SATWIN0 avoid this implicit quantifier alternation by computing such a state in several steps. First, Line 8 computes a state $\mathbf{x}$ and input $\mathbf{i}$ for which *some* $\mathbf{c}$ would make the system leave F. This is a necessary but not a sufficient condition for $\mathbf{x}$ to be a counterexample. Hence, if the query in Line 8 is unsatisfiable, no counterexample can exist, so F must be the final winning region and the algorithm terminates. The formula U in Line 8 excludes state-input combinations which cannot be used by the environment to enforce that F is left.[2] Initially, U is true, i.e., no restrictions are imposed. The refinement of U will be discussed further below. For now, U can be ignored.

If the query in Line 8 is satisfiable, the next step is to check if the candidate $\mathbf{x}$ is indeed a counterexample for the given $\mathbf{i}$. This is investigated in Line 12 by computing some $\mathbf{c}$ for which F is *not* left, i.e., the next state is in F again. If such a $\mathbf{c}$ exists, then the environment cannot enforce that F is left from

[1] This is similar to the procedure SATWIN1 that will be presented in Algorithm 3.4 later. We thus refer to Section 3.1.2.2 for more details.

[2] Formally, U satisfies the invariant $\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big)$.

Algorithm 3.3 SatWin0: Basic SAT solver based CNF learning algorithm for the winning region.

```
 1: procedure SatWin0$\big((\overline{x}, \overline{i}, \overline{c}, I, T, P)\big)$, returns: The winning region $W(\overline{x})$ in CNF or false
 2:     if PropSat$\big(I(\overline{x}) \wedge \neg P(\overline{x})\big)$ then
 3:         return false
 4:     end if
 5:     $F(\overline{x}) := P(\overline{x})$
 6:     $U(\overline{x}, \overline{i}) :=$ true
 7:     while true do
 8:         $(\mathsf{sat}, \mathbf{x}, \mathbf{i}) :=$ PropSatModel$\big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big)$
 9:         if $\neg$sat then
10:             return $F(\overline{x})$
11:         else
12:             $(\mathsf{sat}, \mathbf{c}) :=$ PropSatModel$\big(F(\overline{x}) \wedge \mathbf{x} \wedge \mathbf{i} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big)$
13:             if $\neg$sat then
14:                 $\mathbf{x}_g :=$ PropMinUnsatCore$\big(\mathbf{x}, F(\overline{x}) \wedge \mathbf{i} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big)$
15:                 if PropSat$\big(\mathbf{x}_g \wedge I(\overline{x})\big)$ then
16:                     return false
17:                 end if
18:                 $F(\overline{x}) := F(\overline{x}) \wedge \neg \mathbf{x}_g$
19:                 $U(\overline{x}, \overline{i}) :=$ true
20:             else
21:                 $U := U \wedge \neg$PropMinUnsatCore$\big(\mathbf{x} \wedge \mathbf{i}, \mathbf{c} \wedge F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big)$
22:             end if
23:         end if
24:     end while
25: end procedure
```

state $\mathbf{x}$ with input $\mathbf{i}$. In order to prevent the same $(\mathbf{x}, \mathbf{i})$-pair from being returned by Line 8 again, U could be refined to $U \wedge \neg(\mathbf{x} \wedge \mathbf{i})$. However, by computing the unsatisfiable core of $(\mathbf{x} \wedge \mathbf{i})$ in Line 21, the algorithm may also exclude other $(\mathbf{x}, \mathbf{i})$-pairs for which $\mathbf{c}$ can be used by the system to prevent that F is left. Such $(\mathbf{x}, \mathbf{i})$-pairs are not helpful for the environment in order to enforce that F is left. They can thus safely be removed from U. Note that the formula that is used in the core computation is essentially that of Line 8.

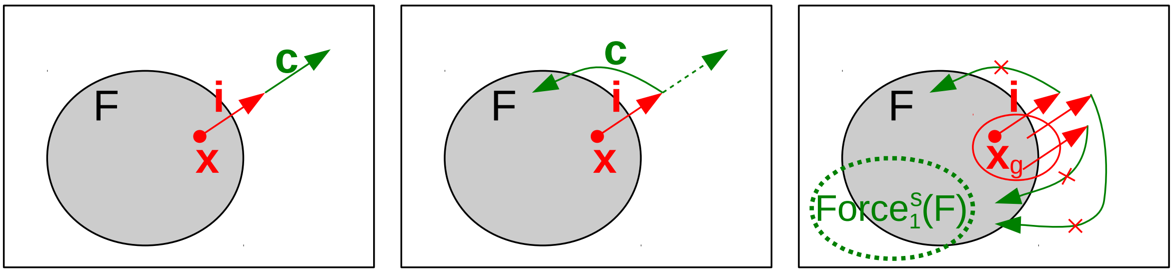
The remaining case is that where the formula in Line 12 is unsatisfiable. In this case, $\mathbf{x}$ is indeed a counterexample because if the environment picks input $\mathbf{i}$, no system action can reach a state of F, so the next state is bound to be in $\neg F$. As for QbfWin, $\mathbf{x}$ cannot be part of the final winning region, so it must be excluded from F. However, before doing so, it is generalized into a larger region $\mathbf{x}_g$ of states that needs to be excluded. This will be explained in the next paragraph. As soon as F changes, U becomes invalid and is thus set to true again in Line 19. The intuitive reason is as follows: even if a certain state-input pair $(\mathbf{x}, \mathbf{i})$ cannot be used by the environment to enforce that F is left, $(\mathbf{x}, \mathbf{i})$ may still be usable for leaving a smaller F because the target region $\neg F(\overline{x}')$ becomes bigger.

Counterexample generalization. QbfWin in Algorithm 3.1 eliminates literals from the counterexample $\mathbf{x}$ as long as the reduced cube $\mathbf{x}_g \subseteq \mathbf{x}$ satisfies $\mathbf{x}_g \wedge F \rightarrow \mathsf{Force}^e_1(\neg F)$, i.e., as long as

$$\exists \overline{x} : \forall \overline{i} : \exists \overline{c}, \overline{x}' : \mathbf{x}_g \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

is unsatisfiable. Due to the universal quantification over the inputs, a SAT solver cannot be used for these checks. SatWin0 solves this issue by considering only one input vector, namely the input $\mathbf{i}$ with which the environment can enforce that F is left from $\mathbf{x}$. For this input $\mathbf{i}$, the formula is certainly unsatisfiable for the full minterm $\mathbf{x}$, because this was checked in Line 12. Hence, eliminating literals from $\mathbf{x}$ while

$$\mathbf{x}_g \wedge \mathbf{i} \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$



(a) Counterexample candidate. **(b)** Counterexample check. **(c)** Generalization.

Figure 3.3: Working principle of SATWIN0. Subfigure (a) illustrates the computation of a counterexample candidate $\mathbf{x}$ such that some $\mathbf{i}, \mathbf{c}$ lead from F to $\neg F$. Subfigure (b) depicts the check if $\mathbf{x}$ is indeed a counterexample by checking if some alternative $\mathbf{c}$ leads back to F. Subfigure (c) illustrates counterexample generalization, enlarging $\mathbf{x}$ into a region $\mathbf{x}_g$ from which input $\mathbf{i}$ enforces that no $\mathbf{c}$ can lead back to F.

is unsatisfiable is implemented in Line 14 by computing an unsatisfiable core of $\mathbf{x}$. Considering only one input vector instead of all makes the formula weaker, which means that less literals may be eliminated. However, the purely propositional satisfiability checks are also potentially faster.

Illustration. Figure 3.3 illustrates the working principle of SATWIN0 graphically. As before, a box represents the set of all states. In Figure 3.3a, a counterexample candidate is computed in form of a state $\mathbf{x}$ from which some input $\mathbf{i}$ and some control value $\mathbf{c}$ lead from F to $\neg F$. This corresponds to the SAT solver call in Line 8. In case of satisfiability, the next step is to check if some alternative $\mathbf{c}$ leads back to F (for the same $\mathbf{x}$ and $\mathbf{i}$). This is illustrated in Figure 3.3b and corresponds to the SAT solver call in Line 12. In case of satisfiability, U is refined in order not to get the same counterexample candidate again (Line 21), and the algorithm proceeds by computing the next counterexample candidate as shown in Figure 3.3a. In case of unsatisfiability, $\mathbf{x}$ is indeed a counterexample. Figure 3.3c illustrates how it is generalized into a larger region $\mathbf{x}_g$ for which input $\mathbf{i}$ enforces that the next state is in $\neg F$: it is ensured that from any state of $\mathbf{x}_g$, with input $\mathbf{i}$, no $\mathbf{c}$ can exist such that the next state is in F again. This is a sufficient but not a necessary condition for $F \wedge \mathbf{x}_g$ not to intersect with $\text{Force}_1^s(F)$. This generalization corresponds to the computation of the unsatisfiable core in Line 14 of SATWIN0. Finally, $\mathbf{x}_g$ is removed from F and the procedure continues with Figure 3.3a, computing the next counterexample candidate.

Discussion. In contrast to QBFWIN (Algorithm 3.1), SATWIN0 potentially requires far more solver calls. This has two reasons. First, many refinements of U may be necessary until a genuine counterexample is found. In contrast, QBFWIN can compute a counterexample with one single solver call. Second, the counterexample generalization in SATWIN0 is weaker and may thus drop fewer literals. This can increase the number of counterexamples that needs to be computed until a solution is found. The advantage of SATWIN0 is that all satisfiability checks are propositional and, thus, potentially less expensive.

The main purpose of discussing the procedure SATWIN0 in Algorithm 3.3 was to prepare for a more advanced version, which will be presented in the next section. Hence, we will not elaborate on implementation aspects or formal correctness arguments for Algorithm 3.3, but only do this for the advanced version, which is presented in the next section.

3.1.2.2 Advanced Algorithm

The basic algorithm from the previous section has two main weaknesses. First, a reset of U needs to be done upon *every* update of F. After such a reset, a lot of iterations may be necessary until U is again restrictive enough for Line 8 to produce a counterexample. Second, incremental solving is difficult in Line 8 due to the negation of F: clauses are added to F, but this makes $\neg F$ weaker, which can only be expressed by (also) removing clauses from the CNF representation of $\neg F$. The procedure SATWIN1 in Algorithm 3.4 resolves these weaknesses. The differences to SATWIN0 are marked in blue.

Algorithm 3.4 SATWIN1: Advanced SAT solver based CNF learning algorithm for the winning region.

```
 1: procedure SATWIN1$\big((\overline{x},\overline{i},\overline{c},I,T,P)\big)$, returns: The winning region $W(\overline{x})$ in CNF or false
 2:     if PROPSAT$\big(I(\overline{x}) \wedge \neg P(\overline{x})\big)$ then
 3:         return false
 4:     end if
 5:     $F(\overline{x}) := P(\overline{x})$
 6:     $U(\overline{x},\overline{i}) :=$ true, $G(\overline{x}) := F(\overline{x})$, precise $:=$ true
 7:     while true do
 8:         $(\mathsf{sat},\mathbf{x},\mathbf{i}) :=$ PROPSATMODEL$\big(F(\overline{x}) \wedge U(\overline{x},\overline{i}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg G(\overline{x}')\big)$
 9:         if $\neg$sat then
10:             if precise then
11:                 return $F(\overline{x})$
12:             end if
13:             $U(\overline{x},\overline{i}) :=$ true, $G(\overline{x}) := F(\overline{x})$, precise $:=$ true
14:         else
15:             $(\mathsf{sat},\mathbf{c}) :=$ PROPSATMODEL$\big(F(\overline{x}) \wedge \mathbf{x} \wedge \mathbf{i} \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge F(\overline{x}')\big)$
16:             if $\neg$sat then
17:                 $\mathbf{x}_g :=$ PROPMINUNSATCORE$\big(\mathbf{x}, F(\overline{x}) \wedge \mathbf{i} \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge F(\overline{x}')\big)$
18:                 if PROPSAT$\big(\mathbf{x}_g \wedge I(\overline{x})\big)$ then
19:                     return false
20:                 end if
21:                 $F(\overline{x}) := F(\overline{x}) \wedge \neg\mathbf{x}_g$
22:                 precise $:=$ false
23:             else
24:                 $U := U \wedge \neg$PROPMINUNSATCORE$\big(\mathbf{x}\wedge\mathbf{i}, \mathbf{c}\wedge F(\overline{x})\wedge U(\overline{x},\overline{i})\wedge T(\overline{x},\overline{i},\overline{c},\overline{x}')\wedge\neg G(\overline{x}')\big)$
25:             end if
26:         end if
27:     end while
28: end procedure
```

Lazy updates of F. The formula $G(\overline{x})$ is a copy of $F(\overline{x})$ that is updated only lazily with newly discovered clauses. Consequently, $F \rightarrow G$ holds at any time, i.e., G always represents a superset of the states in F. The Boolean flag precise is true whenever $G = F$. While SATWIN0 computed a transition from F to $\neg F$ in Line 8, SATWIN1 computes a transition from F to $\neg G$. This is illustrated in Figure 3.4. A transition from F to $\neg G$ is also a transition from F to $\neg F$. Thus, in case of satisfiability, nothing changes. However, if no such transition exists, this does not automatically mean that no transition from F to $\neg F$ exists. Therefore, if $G \neq F$, Line 13 sets $G := F$ and the check is repeated. Only if $G = F$ (indicated by precise $=$ true), the algorithm can conclude that no more counterexample exists and returns F as the final winning region.

Updates of U. New clauses are only added to F but not to G in Line 21. Thus, after any update of F, precise must be set to false. However, U can be kept as it is. The intuitive reason is as follows. If a certain $(\mathbf{x},\mathbf{i})$-pair is not helpful for the environment to enforce a transition from F to $\neg G$, then it will definitely not be helpful to enforce a transition from some smaller set $F \wedge H$ of states into the same region $\neg G$. More formally, we have that

$$\Big((\mathbf{x},\mathbf{i}) \not\models \forall\overline{c} : \exists\overline{x}' : F(\overline{x}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg G(\overline{x}')\Big) \text{ implies } \Big((\mathbf{x},\mathbf{i}) \not\models \forall\overline{c} : \exists\overline{x}' : F(\overline{x}) \wedge H(\overline{x}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg G(\overline{x}')\Big)$$

because

$$\Big(\forall\overline{c} : \exists\overline{x}' : F(\overline{x}) \wedge H(\overline{x}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg G(\overline{x}')\Big) \rightarrow \Big(\forall\overline{c} : \exists\overline{x}' : F(\overline{x}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg G(\overline{x}')\Big).$$

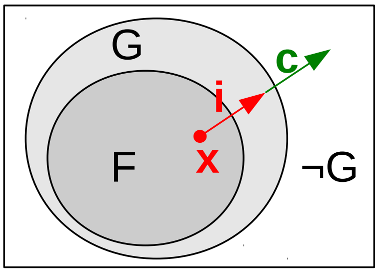


Figure 3.4: Computing a counterexample candidate in SatWin1. While SatWin0 computes a transition from F to $\neg F$, SatWin1 searches for a transition from F to $\neg G$. That is, transitions to $G \wedge \neg F$ are not considered. Since $F \rightarrow G$, this is stricter.

Only when G changes in Line 13, U also becomes invalid and needs to be reset to true.

3.1.2.3 Correctness of the Advanced Algorithm SatWin1

In this section, we will work out a formal correctness argument for SatWin1. In order to increase readability, we split this argument into several lemmas.

Lemma 13. *The* SatWin1 *procedure in Algorithm 3.4 always terminates.*

Proof. Every loop iteration must end with one of the following five events: (1) the loop terminates in Line 11, (2) U is set to true in Line 13, (3) the loop terminates in Line 19, (4) F shrinks in Line 21, or (5) U shrinks in Line 24. We show that all these events lead to termination or eventual shrinking of F: Item (2) cannot happen twice in a row without shrinking F in between: this is prevented by having precise $=$ true. Item (5) cannot happen infinitely often without shrinking F in between because at some point U would reach false, which makes Line 8 return sat $=$ false. In this case, the algorithm either terminates in Line 11, or item (2) occurs, and item (2) cannot occur twice without shrinking F in between. Hence, the loop either terminates or makes some progress towards shrinking F. Before F can shrink below I, the loop definitely terminates in Line 19. □

Lemma 14. SatWin1 *enforces the invariant* $I(\overline{x}) \rightarrow F(\overline{x}) \rightarrow P(\overline{x})$.

Proof. Just as for QbfWin (see Theorem 12), $I \rightarrow F$ is enforced by Line 2 and 18. Likewise, $F \rightarrow P$ is enforced by Line 5 and 21. □

Lemma 15. SatWin1 *enforces the invariant* $W(\overline{x}) \rightarrow F(\overline{x})$.

Proof. Similar to Theorem 12, this can be proven induction: Initially $F = P$, so $W \rightarrow F$ holds because $W \rightarrow P$. Under the hypothesis that $W \rightarrow F$ holds before an update of F in Line 21, it will also hold after the update because Line 17 ensures that

$$\mathbf{x}_g \wedge F(\overline{x}) \wedge \mathbf{i} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

is unsatisfiable. Consequently, we have that

$$\forall \overline{x}, \overline{i}, \overline{c}, \overline{x}' : \big(\mathbf{x}_g \wedge F(\overline{x}) \wedge \mathbf{i}\big) \rightarrow \big(\neg T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \vee \neg F(\overline{x}')\big).$$

Because T is both deterministic and complete ($\overline{x}'$ is always uniquely defined by T; see Definition 5) we can apply the one-point rule (Equation 2.2) in order to rewrite the implication $\forall \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \rightarrow \neg F(\overline{x}')$ to $\exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')$. This gives

$$\forall \overline{x}, \overline{i}, \overline{c} : \exists \overline{x}' : \big(\mathbf{x}_g \wedge F(\overline{x}) \wedge \mathbf{i}\big) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big).$$

Using the one point rule (Equation 2.1) on $\overline{i}$, this formula is equivalent to

$$\forall \overline{x} : \exists \overline{i} : \forall \overline{c} : \exists \overline{x}' : \mathbf{i} \wedge \Big(\big(\mathbf{x}_g \wedge F(\overline{x})\big) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big)\Big).$$

This implies $\forall \overline{x} : \exists \overline{i} : \forall \overline{c} : \exists \overline{x}' : \Big(\big(\mathbf{x}_g \wedge F(\overline{x})\big) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big)\Big)$, which can be written as

$$\mathbf{x}_g \wedge F(\overline{x}) \rightarrow \exists \overline{i} : \forall \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}').$$

By substituting the definition of Force^e_1, we get $\mathbf{x}_g \wedge F \rightarrow \mathsf{Force}^e_1(\neg F)$. Using the induction hypothesis $W \rightarrow F$, which can be written as $\neg F \rightarrow \neg W$, this means that $\mathbf{x}_g \wedge F \rightarrow \mathsf{Force}^e_1(\neg W)$ holds. Thus, only states that cannot be part of W are removed in Line 21. In other words, F cannot shrink below W. □

The following lemma states that the formula $F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})$ can only represent state-input pairs for which the system player can reach G and thus avoid ending up in $\neg G$. In other words, the conjunction with U in the SAT solver call of Line 8 excludes only state-input pairs for which the environment cannot enforce a transition from F to $\neg G$.

Lemma 16. SATWIN1 *enforces the invariant* $\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big)$.

Proof. U is initialized to true, so the invariant holds initially. Line 13 retains the invariant because U is set to true. Line 21 retains the invariant because F can only get stricter. It remains to be shown that Line 24 retains the invariant. Let $\mathbf{u}$ be the result of PROPMINUNSATCORE in Line 24. The update $U := U \wedge \neg \mathbf{u}$ in Line 24 changes the invariant to

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge (\neg U(\overline{x}, \overline{i}) \vee \mathbf{u})\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big),$$

which can be written as

$$\forall \overline{x}, \overline{i} : \Big(\big(F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})\big) \vee \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge \mathbf{u}\big) \Big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big).$$

In general, a formula $(A \vee B) \rightarrow C$ holds iff $A \rightarrow C$ and $B \rightarrow C$. By induction, we already know that $\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big)$ holds. Hence, what remains to be shown is that $\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge \mathbf{u}\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big)$ also holds. Since $\mathbf{u} \wedge \mathbf{c} \wedge F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$ is unsatisfiable (enforced by Line 24), we have that

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge \mathbf{u}\big) \rightarrow \big(\forall \overline{c}, \overline{x}' : \neg \mathbf{c} \vee \neg T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \vee G(\overline{x}')\big).$$

By applying the one-point rule (Equation 2.1 for $\overline{c}$ and Equation 2.2 for $\overline{x}'$), this can also be written as

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge \mathbf{u}\big) \rightarrow \big(\exists \overline{c}, \overline{x}' : \mathbf{c} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big).$$

This formula obviously implies

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge \mathbf{u}\big) \rightarrow \big(\exists \overline{c}, \overline{x} : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge G(\overline{x}')\big),$$

which was to be shown for Line 24 to preserve the invariant. □

Lemma 17. *If* SATWIN1 *reaches Line 11,* $F(\overline{x}) = W(\overline{x})$ *holds at that point.*

Proof. Line 11 is only reached when $G = F$ (otherwise $\mathsf{precise}$ is false) and

$$F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$$

is unsatisfiable, which means that

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i})\big) \rightarrow \big(\forall \overline{c} : \forall \overline{x}' : \neg T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \vee F(\overline{x}')\big)$$

holds. By applying the one-point rule (Equation 2.2), this can also be written as

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i})\big) \rightarrow \big(\forall \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big).$$

In turn, this implies

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge U(\overline{x}, \overline{i})\big) \rightarrow \big(\exists \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big).$$

From Lemma 16, we know that

$$\forall \overline{x}, \overline{i} : \big(F(\overline{x}) \wedge \neg U(\overline{x}, \overline{i})\big) \rightarrow \big(\exists \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big).$$

Since $A \wedge B \rightarrow C$ and $A \wedge \neg B \rightarrow C$ together imply $A \rightarrow C$, we can conclude that

$$\forall \overline{x} : F(\overline{x}) \rightarrow \forall \overline{i} : \exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

must hold in Line 11. This means that the returned F satisfies $F \rightarrow \mathsf{Force}_1^s(F)$. From $W \rightarrow F \rightarrow P$ (Lemma 15 and 14), it follows that $F = W$. The reason is that W is the set of *all* states from which the system player can enforce the specification, i.e., no proper superset H of W can satisfy $H \rightarrow P$ and $H \rightarrow \mathsf{Force}_1^s(H)$. □

Theorem 18. *The* SATWIN1 *procedure in Algorithm 3.4 returns the winning region $W(\overline{x})$ of a given safety specification $\mathcal{S}$, or* false *if the specification is unrealizable.*

Proof. Unrealizability: If $\mathcal{S}$ is unrealizable, $I \not\rightarrow W$. SATWIN1 terminates (Lemma 13), but cannot terminate in Line 11 because $F = W$ (Lemma 17) contradicts with $I \not\rightarrow W$ (unrealizability) and $I \rightarrow F$ (Lemma 14). Hence, in case of unrealizability, SATWIN1 must terminate in Line 3 or 19 returning false.

Realizability: SATWIN1 can only return false in Line 3 or 19 if F is about to be updated in such a way that $I \not\rightarrow F$. However, from $I \rightarrow W$ (realizability) and $W \rightarrow F$ (Lemma 15), it follows that $I \rightarrow F$, so this can never happen. Yet, Lemma 13 says that SATWIN1 terminates, so it must reach Line 11 eventually. By Lemma 17, this will return the winning region. □

3.1.2.4 Efficient Implementation

This section discusses some important aspects of implementing SATWIN1 efficiently.

Incremental solving. We propose to use three SAT solver instances incrementally. The first one will be called solverC and stores $F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$. solverC is used in Line 8 and Line 24, where the conjunction with $\mathbf{c}$ is realized with temporarily asserted assumption literals. Whenever Line 13 is reached, solverC is reset with the new CNF encoding of $\neg G(\overline{x}') = \neg F(\overline{x}')$. Otherwise, clauses are only added to F or U. The second solver instance, called solverG, stores $F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$ and is used for Line 15 and Line 17. Clauses are only added to F, so solverG does not have to be reset at all. The conjunctions with $\mathbf{i}$ and $\mathbf{x}$, which change from iteration to iteration, are again realized by setting assumption literals. The lines 15 and 17 are actually combined into one SAT solver call that returns either a satisfying assignment $\mathbf{c}$ or an unsatisfiable core. The third solver instance stores $I(\overline{x})$ and is used in Line 18.[3] The conjunction with $\mathbf{x}_g$ is again realized with assumption literals.

CNF compression. Whenever solverC is reset with the current CNF encoding of $\neg G(\overline{x}') = \neg F(\overline{x}')$ in Line 13, we call the procedure COMPRESSCNF from Algorithm 3.2 (with literal dropping disabled) in order to reduce the size of F beforehand. This results in a more compact CNF encoding of $\neg G(\overline{x}')$ when using the method of Plaisted and Greenbaum [174].

Resets of solverG. By default, we only add clauses to solverG. However, after some iterations, many of the F-clauses added to solverG can become redundant because they can be implied by (a

[3] The input format in our implementation actually allows for only one initial state, so the check in Line 18 can be realized syntactically without calling a SAT solver.

combination of) other clauses that have been added later. To prevent the clause database of solverG from growing unreasonably, we also reset solverG with the compressed F from time to time. As a heuristic, we track the number of F-clauses that have been added to solverG so far, and compute the difference to the number of clauses in the compressed F. If this difference exceeds a certain limit, solverG is reset. This can give a moderate speedup for certain SAT solvers and benchmarks.

3.1.3 Partial Quantifier Expansion

The procedure QBFWIN in Algorithm 3.1 uses quantified formulas to compute counterexamples witnessing that $F \neq W$ and to generalize these counterexamples. In contrast, the procedure SATWIN1 in Algorithm 3.4 avoids the universal quantifiers. This results in less expensive solver calls, but comes at the price of requiring more iterations of the outer loop. In this section, we will discuss a hybrid approach which quantifies universally over *some* (but not necessarily all) variables. The universal quantification is then eliminated by applying universal expansion so that the resulting formulas can be solved with a plain SAT solver. The hope is to find a sweet spot where the reduction in the number of iterations is more significant than the additional costs per solver call.

3.1.3.1 Quantifier Expansion in Counterexample Computation

The procedure QBFWIN in Algorithm 3.1 computes counterexamples to $F = W$ by solving the quantified formula

$$\exists \overline{x}, \overline{i} : \forall \overline{c} : \exists \overline{x}' : F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}').$$

In contrast, the SATWIN1 procedure from Algorithm 3.4 avoids the universal quantification of the variables $\overline{c}$ by solving the formula

$$\exists \overline{x}, \overline{i} : \exists \overline{c} : \exists \overline{x}' : F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}'),$$

where G is just a copy of F that may not be fully up to date. The latter formula does not necessarily yield a counterexample, but only a candidate. If the candidate turns out to be spurious, it is excluded by refining U. This approach can be seen as a "lazy elimination" of the universal quantification over $\overline{c}$ via U. The disadvantage is that many refinements of U may be necessary before the first genuine counterexample is found. One alternative would be to eliminate $\forall \overline{c}$ in

$$\exists \overline{x}, \overline{i} : \forall \overline{c} : \exists \overline{x}' : F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$$

eagerly by performing universal expansion as explained in Section 2.2.2. Yet, this may blow up the formula size by a factor of $2^{|\overline{c}|}$ and may thus be infeasible. Another alternative is to partition the variables of $\overline{c}$ into two subsets $\overline{c}_1$ and $\overline{c}_2$ and solve

$$\exists \overline{x}, \overline{i} : \exists \overline{c}_1 : \forall \overline{c}_2 : \exists \overline{x}' : F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$$

using a SAT solver by expanding only over the variables in $\overline{c}_2$. By adjusting the relative size of $\overline{c}_2$, different trade-offs between decreasing the number of refinements to U and increasing the costs per solver call can be achieved.

3.1.3.2 Quantifier Expansion in Counterexample Generalization

The idea is similar to that of the previous subsection. QBFWIN eliminates literals from a counterexample $\mathbf{x}$ as long as

$$\exists \overline{x} : \forall \overline{i} : \exists \overline{c}, \overline{x}' : \mathbf{x}_g \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

is unsatisfiable. In contrast, SATWIN1 avoids the universal quantification over $\overline{i}$ by ensuring that

$$\exists \overline{x} : \exists \overline{i} : \exists \overline{c}, \overline{x}' : \mathbf{x} \wedge \mathbf{i} \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

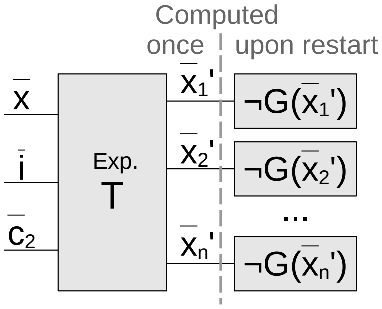


Figure 3.5: Universal expansion for counterexample computation. The expansion of the transition relation T is computed only once. The expansion of $\neg G(\overline{x}')$ needs to be recomputed upon every solver restart.

is unsatisfiable for some concrete **i**. The latter check is potentially cheaper, but may result in fewer literals being eliminated from $\mathbf{x}$. This means that the refinement of F is less substantial, so more iterations may be needed. By partitioning the variables $\overline{i}$ into $\overline{i}_1$ and $\overline{i}_2$ and checking

$$\exists \overline{x} : \exists \overline{i} : \mathbf{x} \wedge \mathbf{i} \wedge F(\overline{x}) \wedge \forall \overline{i}_2 : \exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$$

for unsatisfiability, different trade-offs between the generalization procedure of QBFWIN and that of SATWIN1 can be realized.

3.1.3.3 Efficient Implementation

Universal expansion needs to be implemented carefully in order to avoid an unnecessary blow-up of the formula size, and to keep the time for the expansion low. Our experience showed that even small inefficiencies can cost orders of magnitude in both metrics.

Expansion for counterexample computation. Since $F(\overline{x})$ and $U(\overline{x}, \overline{i})$ are independent of $\overline{c}$ and $\overline{x}'$, we only apply universal expansion to

$$\forall \overline{c}_2 : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg G(\overline{x}')$$

and conjoin $F(\overline{x}) \wedge U(\overline{x}, \overline{i})$ afterwards. The transition relation T is always fixed, but $\neg G(\overline{x}')$ changes upon every restart of solverC in Line 13 of SATWIN1. Hence, we expand T only once and store the resulting renamings $\overline{x}_1', \ldots \overline{x}_n'$ of $\overline{x}'$. A copy of $\neg G(\overline{x}')$ is then added for each renaming $\overline{x}_i'$ when solverC is initialized or restarted. This is illustrated in Figure 3.5.

Expansion of T. In our implementation, T is originally given as a circuit of AND-gates (where inputs can be negated). We perform the expansion of T directly on this circuit and only encode the result into CNF. This facilitates efficient constant propagation and other simplifications. When expanding a certain $c \in \overline{c}_2$, we only copy those AND-gates that have c in their fan-in cone. Whenever the copy of some AND-gate has the same inputs as some existing AND-gate, the existing gate is reused. Finally, the tool ABC [42] is called to simplify the expanded circuit. This involves fraiging, which ensures that no two nodes in the circuit can represent the same function over the inputs. Hence, equivalent (copies of) next-state signals will be represented by the same variable, which enables a more substantial simplification of the $\neg G(\overline{x}')$ copies (see Figure 3.5). Finally, duplicate renamings of the next-state variables are removed. Since T is expanded only once, these simplifications can be afforded.

Expansion of $\neg G(\overline{x}')$. First, we perform an even more aggressive compression of $G(\overline{x}')$ than done by COMPRESSCNF in Algorithm 3.2. COMPRESSCNF removes a clause c from a CNF A if $\big(A \setminus \{c\}\big) \rightarrow c$, i.e., if the clause is implied by other clauses of A already. We now remove a clause c from $G(\overline{x}')$ if

$$\Big(F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(G(\overline{x}') \setminus \{c\}\big)\Big) \rightarrow c.$$

Hence, the compressed $G(\overline{x}')$ will only be equivalent to the original $G(\overline{x}')$ if the current state is in F, but this is asserted in all SAT solver calls of SATWIN1 anyway. For every renaming $\overline{x}_i'$ of $\overline{x}'$ that has been created during the expansion of T, we then perform the following steps: First, $G(\overline{x}_i')$ is computed by applying the renaming. Second, $G(\overline{x}_i')$ is simplified by removing tautological clauses and performing unit clause propagation. Finally, $G(\overline{x}_i')$ is negated, while auxiliary variables that have already been introduced during the negation of other copies of $G(\overline{x}')$ are reused. All these measures contribute towards reducing the size of the expansion of $\neg G(\overline{x}')$.

Expansion in counterexample generalization. This is easier, since no negation of a CNF is involved. Again, T is expanded and the resulting renamings $\overline{x}_1', \dots \overline{x}_n'$ of $\overline{x}'$ are stored. Whenever a clause $\neg\mathbf{x}_g$ is added to F, we do not only add it to solverG but also add all the renamed next-state copies of the clause to solverG.

Configuration. In our experiments for expansion in counterexample computation, choosing low numbers for $|\overline{c}_2|$ only slowed down the SATWIN1 procedure compared to $|\overline{c}_2| = 0$. High numbers for $|\overline{c}_2|$ did bring a speedup, though, with the best results achieved for $\overline{c}_2 = \overline{c}$. Furthermore, we observed that an explosion of the formula size can be avoided in most cases by our careful implementation of the formula expansion. Hence, by default, we expand over all variables in $\overline{c}$ and only fall back to $\overline{c}_2 = \emptyset$ if some memory limit is exceeded. For counterexample generalization, a speedup could only be achieved with low numbers of $|\overline{i}_2|$. Hence, by default, we only expand one input signal. As a heuristic, the signal that causes the least number of gates to be copied when expanding the transition relation is chosen.

Discussion. Our optimization of partial quantifier expansion can be used to realize different trade-offs between the number of SAT solver calls and their costs in Algorithm 3.4. We hoped to find a sweet spot between these two cost factors at low expansion rates, but our experiments suggest high rates at least for counterexample computation. While the basic idea of quantifier expansion is simple, such high expansion rates require a careful implementation, like the one discussed in this section, in order not to waste computational resources.

3.1.4 Reachability Optimizations

In this section, we present optimizations that exploit (un)reachability information when computing a winning region with query learning. The optimizations can be applied to QBFWIN (Algorithm 3.1) and to SATWIN1 (Algorithm 3.4), both with and without partial quantifier elimination. However, to simplify the presentation, we only explain the optimizations for the case of QBFWIN in detail. The application to SATWIN1 works in exactly the same way.

3.1.4.1 Optimization RG: Reachability for Counterexample Generalization

Recall that a counterexample $\mathbf{x} \models F \wedge \mathsf{Force}_1^e(\neg F)$ in QBFWIN is a state that is part of the current over-approximation F of the winning region, but this state cannot be part of the final winning region. The state is represented by a minterm $\mathbf{x}$ over the state variables $\overline{x}$. QBFWIN generalizes the counterexample $\mathbf{x}$ into a larger state region $\mathbf{x}_g$ by eliminating literals as long as $F \wedge \mathbf{x}_g \rightarrow \mathsf{Force}_1^e(\neg F)$ holds, i.e., as long as $F \wedge \mathbf{x}_g \wedge \mathsf{Force}_1^s(F)$ is unsatisfiable. The reason is that any state $\mathbf{x}_a \models F \wedge \mathbf{x}_g \wedge \mathsf{Force}_1^s(F)$ could potentially be part of the winning region, and thus must not be removed from F. Yet, as an optimization, we can still remove such a state $\mathbf{x}_a$, as long as it is guaranteed that $\mathbf{x}_a$ is unreachable from the initial states. Using this insight, we can eliminate literals in a counterexample $\mathbf{x}$ as long as $R \wedge F \wedge \mathbf{x}_g \rightarrow \mathsf{Force}_1^e(\neg F)$, where $R(\overline{x})$ is an over-approximation of the reachable states in $\mathcal{S}$. In QBFWIN, this can be realized by conjoining R to the QBF that is checked in Line 12. This may result in more literals being eliminated during the generalization, which means that F is pruned more extensively. Ultimately, this can reduce the number of iterations in QBFWIN.

Computing reachable states. The states that are reachable from the initial states in a specification $\mathcal{S}$ can be defined inductively as follows: All states in $I(\overline{x})$ are reachable. If a state $\mathbf{x}$ is reachable, then

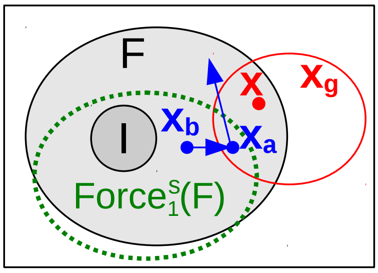


Figure 3.6: Optimization RG: An overlap of $F \wedge \mathbf{x}_g$ with $\mathsf{Force}_1^s(F)$ is allowed as long as no state $\mathbf{x}_a \models F \wedge \mathbf{x}_g \wedge \mathsf{Force}_1^s(F)$ is initial or has a predecessor $\mathbf{x}_b$ in $F \wedge \neg\mathbf{x}_g$. Thus, the depicted generalization $\mathbf{x}_g$ would not be allowed.

all states $\mathbf{x}' \models \exists \overline{x}, \overline{i}, \overline{c} : \mathbf{x} \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ are also reachable. In the synthesis setting, this definition can even be refined. Any over-approximation F of the winning region is itself an over-approximation of the reachable states, not necessarily in the specification $\mathcal{S}$, but definitely in the final implementation. The reason is that no realization of $\mathcal{S}$ must ever leave the winning region W, and thus also not F. This insight can be used to compute an even tighter set of reachable states by considering only transitions that remain in F. In principle, the set of reachable states can easily be computed using a simple fixed-point algorithm. However, we consider this to be too expensive, and instead work with over-approximations of the reachable states. Such over-approximations are also useful in formal verification, and many methods to compute them exist [161].

Our approach. We avoid computing an over-approximation of the reachable states explicitly. Instead, we use an idea that is inspired by the model checking algorithm IC3 [41]: Let $R(\overline{x})$ be an over-approximation of the reachable states. By induction, we know that a state $\mathbf{x}$ is definitely unreachable if $I(\overline{x}) \rightarrow \neg\mathbf{x}$ and $\neg\mathbf{x} \wedge R(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \rightarrow \neg\mathbf{x}'$. The formula says that if the current state is reachable but different from $\mathbf{x}$, then the next state cannot be $\mathbf{x}$ either. Hence, if $\mathbf{x}$ is not an initial state, then $\mathbf{x}$ can never be visited. The same reasoning applies if $\mathbf{x}$ is an incomplete cube (or any other formula) representing a set of states. In IC3, $\neg\mathbf{x}$ is said to be *inductive relative to* the current knowledge R about the reachable states. It can thus be used to refine R.

In our synthesis setting, we take the current over-approximation F as an over-approximation of the reachable states. When generalizing a counterexample $\mathbf{x}$, literals cannot only be eliminated if $F \wedge \mathbf{x}_g \rightarrow \mathsf{Force}_1^e(\neg F)$ is preserved, but also if $\neg\mathbf{x}_g$ is inductive relative to F. The two criteria can be combined by requiring that

$$\exists \overline{x}^*, \overline{i}^*, \overline{c}^*, \overline{x} : \forall \overline{i} : \exists \overline{c}, \overline{x}' : \big(I(\overline{x}) \vee F(\overline{x}^*) \wedge \neg\mathbf{x}_g^* \wedge T(\overline{x}^*, \overline{i}^*, \overline{c}^*, \overline{x})\big) \wedge \\ \mathbf{x}_g \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}') \quad (3.1)$$

is unsatisfiable. Only the parts of the formula that are marked in blue are new. The variables $\overline{x}^*$, $\overline{i}^*$ and $\overline{c}^*$ are previous-state copies of $\overline{x}$, $\overline{i}$ and $\overline{c}$, respectively. The original version of the formula was true if some state $\mathbf{x}_a \models F \wedge \mathbf{x}_g \wedge \mathsf{Force}_1^s(F)$ exists. The improved formula also requires that $\mathbf{x}_a$ is either an initial state, or has a predecessor $\mathbf{x}_b$ in $F \wedge \neg\mathbf{x}_g$. This is illustrated in Figure 3.6. If neither of these two criteria holds, then we know that $I(\overline{x}) \rightarrow \neg\mathbf{x}_a$ and $\neg\mathbf{x}_a \wedge F(\overline{x}) \wedge \mathbf{x}_g \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \rightarrow \neg\mathbf{x}_a'$. This means that $\neg\mathbf{x}_a$ is inductive relative to $F \wedge \neg\mathbf{x}_g$, so $\mathbf{x}_a$ is unreachable and can thus be removed even if it could potentially be part of the winning region. Note that we do not require inductiveness relative to F but rather relative to $F \wedge \neg\mathbf{x}_g$. The intuitive reason is that F will be updated to $F \wedge \neg\mathbf{x}_g$, so a predecessor $\mathbf{x}_b$ in $F \wedge \mathbf{x}_g$ does not count. The following theorem states that this procedure cannot prune F too much.

Theorem 19. *For a realizable specification $\mathcal{S}$, if Equation 3.1 is unsatisfiable, then $F \wedge \mathbf{x}_g$ cannot contain a state $\mathbf{x}_a$ from which (a) the system player can enforce that F is visited in one step, and (b) which is reachable in any implementation of $\mathcal{S}$.*

Proof. By contradiction, assume that there exists such as state $\mathbf{x}_a$. Any implementation of $\mathcal{S}$ must only

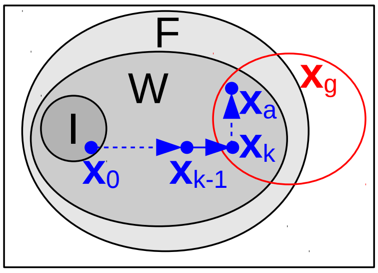


Figure 3.7: Illustration of the correctness proof for optimization RG: $\mathbf{x}_0, \dots, \mathbf{x}_n$ is a path of states from I to $\mathbf{x}_a$. The entire path is contained in W, which is a subset of F. From $\mathbf{x}_k$ on, the path remains in $F \wedge \mathbf{x}_g$ until $\mathbf{x}_a$ is reached.

visit states in W. Hence, for $\mathbf{x}_a$ to be reachable in the final implementation, there must exist a play $\mathbf{x}_0, \dots, \mathbf{x}_n, \dots$ of the game $\mathcal{S}$ such that

- $\mathbf{x}_0 \models I$ (the play starts in the initial states),
- $\mathbf{x}_n = \mathbf{x}_a$ (the play reaches $\mathbf{x}_a$ at some step n),
- $n \geq 1$ (that is, $\mathbf{x}_a$ cannot be initial because this would satisfy Equation 3.1), and
- $\mathbf{x}_j \models W$ for all $0 \leq j \leq n$ (all states in the path from I to $\mathbf{x}_a$ are in the winning region, and thus potentially reachable).

Such a play is illustrated in Figure 3.7. Since $\mathbf{x}_a \models F \wedge \mathbf{x}_g$, there must exist a smallest $k \leq n$ such that $\mathbf{x}_j \models F \wedge \mathbf{x}_g$ for all $k \leq j \leq n$. Now, $\mathbf{x}_k$ is either initial or has a predecessor $\mathbf{x}_{k-1}$ in $F \wedge \neg\mathbf{x}_g$ (because $\mathbf{x}_{k-1} \models W, W \rightarrow F$, and $\mathbf{x}_{k-1} \not\models F \wedge \mathbf{x}_g$). Thus, $\mathbf{x}_k$ satisfies the new (blue) part of Equation 3.1. Since Equation 3.1 is unsatisfiable, the system player cannot enforce that the play traverses from $\mathbf{x}_k$ to F. Hence, $\mathbf{x}_k$ cannot be part of W. This contradiction means that such a path of reachable states ending in $\mathbf{x}_a$ cannot exist if Equation 3.1 is unsatisfiable. □

Theorem 19 only considers the case of a realizable specification. In case of unrealizability, the correctness argument is even simpler: Optimization RG cannot make QBFWIN identify an unrealizable specification as realizable because the additional conjuncts in Equation 3.1 can only have the effect that *more* states are removed from F, thus F can only shrink below I faster. Another important remark is that QBFWIN no longer computes the winning region when optimization RG is enabled, but only a winning area according to Definition 11. The reason is that states of W may be missing in F if they are unreachable.

3.1.4.2 Optimization RC: Reachability for Counterexample Computation

Similar to improving the generalization of counterexamples using unreachability information, we can also restrict their computation to potentially reachable states. In addition to $\mathbf{x} \models F \wedge \mathsf{Force}_1^e(\neg F)$, we require that the counterexample $\mathbf{x}$ is either an initial state, or has a predecessor in F that is different from $\mathbf{x}$. If neither of these two conditions is satisfied, then $\mathbf{x}$ can only be unreachable and, thus, does not have to be removed from F.

Realization. In QBFWIN, these additional constraints can be imposed by modifying the QBF query in Line 7 to

$$(\mathsf{sat}, \mathbf{x}) := \textsc{QbfSatModel}\big(\exists \overline{x}^*, \overline{i}^*, \overline{c}^*, \overline{x}, \overline{i} : \forall \overline{c} : \exists \overline{x}' : \\ \big(I(\overline{x}) \vee \overline{x}^* \neq \overline{x} \wedge F(\overline{x}^*) \wedge T(\overline{x}^*, \overline{i}^*, \overline{c}^*, \overline{x})\big) \wedge F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')\big). \quad (3.2)$$

As before, the new parts are marked in blue, and $\overline{x}^*$, $\overline{i}^*$, and $\overline{c}^*$ are the previous-state copies of $\overline{x}$, $\overline{i}$, and $\overline{c}$, respectively. The expression $\overline{x}^* \neq \overline{x}$ encodes that at least one state variable $x \in \overline{x}$ must have a different value than its corresponding previous-state copy.

Consequences. When executing LEARNQBF with optimization RC on a realizable specification, the returned formula F may not be a winning area according to Definition 11 any more. More specifically, item (3) of Definition 11 may be violated because from some unreachable states of F, it may be that the system player cannot enforce that F is reached in the next step. Consequently, a system implementation can no longer be computed as a Skolem function for the variables $\overline{c}$ in the formula

$$\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(F(\overline{x}) \rightarrow F(\overline{x}')\big)$$

because this formula no longer holds true. Still, a system implementation can be extracted, e.g., by computing Skolem functions for the $\overline{c}$-signals in the negation of Equation 3.2.

Configuration. In our experiments, we achieve a significant speedup when applying optimization RG, especially with our SAT solver based algorithm SATWIN1. Optimization RC also gives some speedup for certain benchmarks, but does not pay off on average. Hence, by default, we apply optimization RG but disable optimization RC.

3.1.5 Template-Based Approach

In the previous sections, a winning area was computed iteratively by starting with some initial approximation and then refining this approximation based on counterexamples. This section presents a completely different approach, where we simply assert the constraints that constitute a winning area and compute a solution in one go.

Basic idea. We define a generic template $H(\overline{x}, \overline{k})$ for the winning area $F(\overline{x})$ we wish to construct. Here, $H(\overline{x}, \overline{k})$ is a formula over the state variables $\overline{x}$ and a vector of Boolean variables $\overline{k}$, which act as template parameters. Concrete values $\mathbf{k}$ for the parameters $\overline{k}$ instantiate a concrete formula $F(\overline{x}) = H(\overline{x}, \mathbf{k})$ over the state variables $\overline{x}$. This reduces the search for a propositional formula (the winning area) to a search for Boolean template parameter values. We can now compute a winning area according to Definition 11 with a single QBF solver call, simply by asserting the desired properties:

$$\begin{aligned}(sat, \mathbf{k}) = \textsc{QbfSatModel}\Big(\exists \overline{k} : \forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : &\big(I(\overline{x}) \rightarrow H(\overline{x}, \overline{k})\big) \wedge \\ &\big(H(\overline{x}, \overline{k}) \rightarrow P(\overline{x})\big) \wedge \\ &\big(H(\overline{x}, \overline{k}) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge H(\overline{x}', \overline{k})\big)\big)\Big)\end{aligned} \quad (3.3)$$

With the resulting template parameter values $\mathbf{k}$, the corresponding instantiation $F(\overline{x}) = H(\overline{x}, \mathbf{k})$ of $H(\overline{x}, \overline{k})$ can then be computed.

Completeness of templates. A template $H(\overline{x}, \overline{k})$ does not necessarily have to be complete in the sense that it can represent *every* function $F(\overline{x})$ over the state variables with some choice for the parameters $\overline{k}$. We rather restrict the expressiveness of templates deliberately in order to reduce the search space for the solver. The underlying assumption is that many specifications have a winning area which can be represented as a "simple" formula over the state variables (according to some metric). We will use templates that are parameterized in their expressive power. As a general strategy, we will start with a low value for some expressiveness parameter N, and increase N as long as Equation 3.3 is unsatisfiable. Detecting unrealizability is difficult with this approach, though. Only if Equation 3.3 is unsatisfiable for a template that can represent *every* function $F(\overline{x})$ over the state variables, we can conclude that the corresponding specification is unrealizable.

Concrete realizations. While the basic idea of the template-based approach is simple, there are many ways to realize it. One degree of freedom lies in the definition of the generic template $H(\overline{x}, \overline{k})$ and its parameters. Two concrete suggestions will be made in the following subsections. Another source of freedom lies in the way to solve Equation 3.3. An approach using SAT solvers instead of a single call to a QBF solver will be presented in Section 3.1.5.3.

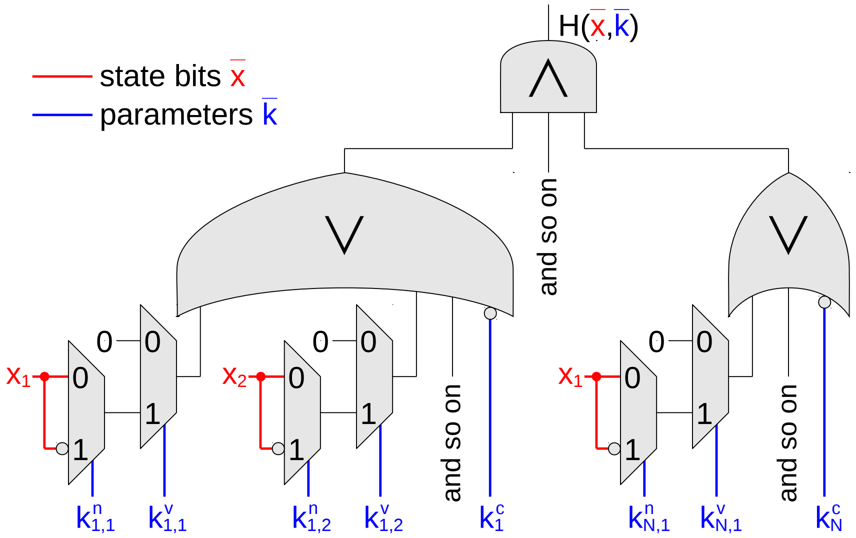


Figure 3.8: Circuit illustration of a generic CNF template $H(\overline{x}, \overline{k})$. Template parameters $\overline{k}$ are marked in blue, state variables $\overline{x}$ are marked in red. The truth constants true and false are abbreviated by 1 and 0, respectively. The parameters k^c_i define if clause i is used. The parameters $k^v_{i,j}$ define if clause i contains variable x_j or not. The parameters $k^n_{i,j}$ define if x_j appears negated or unnegated in clause i.

3.1.5.1 CNF Templates

Figure 3.8 shows a circuit that illustrates how the template $H(\overline{x}, \overline{k})$ can be defined as a parameterized CNF formula over the state variables $\overline{x}$. That is, $F(\overline{x})$ is represented as a conjunction of clauses over the state variables. Template parameters $\overline{k}$ define the shape of the clauses. The trapezoids in Figure 3.8 are multiplexers that select one of the inputs on the left depending on the value of the signal fed in from below. A CNF encoding of this circuit such that it can be used in Equation 3.3 is straightforward [208].

The construction in Figure 3.8 works as follows. First, a maximum number N of clauses is fixed. This number configures the expressiveness of the template. Next, three vectors $\overline{k^c}$, $\overline{k^v}$, $\overline{k^n}$ of template parameters are introduced. Together, they form $\overline{k} = \overline{k^c} \cup \overline{k^v} \cup \overline{k^n}$. The meaning of the parameters is as follows.

- If parameter k^c_i with $1 \leq i \leq N$ is true, then clause i is used in $F(\overline{x})$, otherwise not. This is achieved by making the clause true (and thus irrelevant in the conjunction of clauses) if k^c_i is false.
- If parameter $k^v_{i,j}$ with $1 \leq i \leq N$ and $1 \leq j \leq |\overline{x}|$ is true, then the state variable $x_j \in \overline{x}$ appears in clause i of $F(\overline{x})$, otherwise not. This is realized with a multiplexer that sets the corresponding literal in the clause to false (thus making it irrelevant in the disjunction) if $k^v_{i,j}$ is false.
- Finally, if parameter $k^n_{i,j}$ is true, then the state variable x_j can appear in clause i only negated, otherwise only unnegated. This is realized with a multiplexer that selects between x_j and $\neg x_j$. If $k^v_{i,j}$ is false, then $k^n_{i,j}$ is irrelevant.

This results in $|\overline{k}| = 2 \cdot N \cdot |\overline{x}| + N$ template parameters.

Example 20. For $\overline{x} = (x_1, x_2, x_3)$ and $N = 3$, the CNF $(x_1 \vee \neg x_2) \wedge (\neg x_3)$ can be realized with

- $k^c_1 = k^c_2 =$ true and $k^c_3 =$ false (only clause 1 and 2 are used),
- $k^v_{1,1} = k^v_{1,2} =$ true and $k^v_{1,3} =$ false (clause 1 contains x_1 and x_2 but not x_3),
- $k^v_{2,3} =$ true and $k^v_{2,1} = k^v_{2,2} =$ false (clause 2 contains x_3 but not x_1 and not x_2),
- $k^n_{1,1} =$ false and $k^n_{1,2} =$ true (clause 1 contains x_1 unnegated and x_2 negated), and
- $k^n_{2,3} =$ true (clause 2 contains x_3 negated).

All other parameters are irrelevant.

Choosing N is delicate. If N is too low, we will not find a solution, even if one exists. If it is too high, we waste computational resources and may find an unnecessarily complex winning region. In our

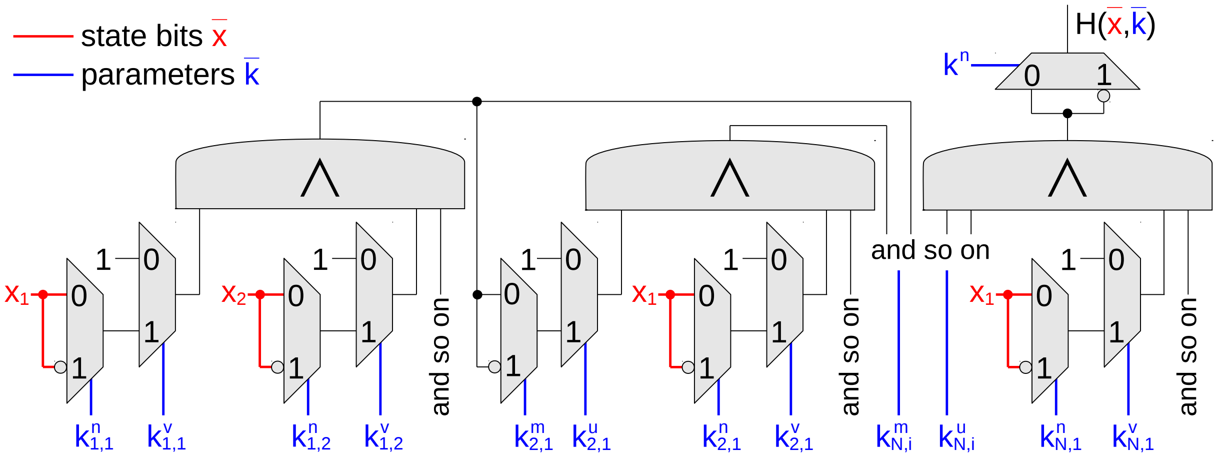


Figure 3.9: Circuit illustration of a generic AND-inverter graph template $H(\overline{x}, \overline{k})$. The truth constants true and false are again abbreviated by 1 and 0, respectively. Template parameters $\overline{k}$ (marked in blue) define if a certain state variable x_i (marked in red) appears as input of a gate or not, and if it appears negated or not. Other parameters define if outputs of a previous gate appear as input, and if negated or not.

implementation, we solve this dilemma by starting with $N = 1$ and increasing N by one upon failure until we reach $N = 4$. From there, we double N upon failure. We stop if we get a negative answer for $N \geq 2^{|\overline{x}|}$ because any Boolean formula over $\overline{x}$ can be represented in a CNF with less than $2^{|\overline{x}|}$ clauses.

3.1.5.2 AND-Inverter Graph Templates

Another option is to define the template $H(\overline{x}, \overline{k})$ as a network of AND-gates and inverters, fed by the state variables $\overline{x}$. The template parameters $\overline{k}$ define the connections between the gates and the state variables, as well as the negation of signals.

Figure 3.9 gives a concrete proposal for defining such a template. The template is again illustrated as a circuit, but can easily be encoded into CNF. A maximum number N of AND-gates is chosen first. The first gate can have all state variables as input, either negated or unnegated. The second gate can also have the output of the first gate as input. The third gate can have the output of the first two gates as additional inputs, and so on. The output of the last AND-gate defines $H(\overline{x}, \overline{k})$, again with a possible negation. The template parameters $\overline{k}$ define which inputs of a gates are actually used or ignored, and which inputs are used negated or unnegated. We distinguish five groups of parameters.

- If parameter $k^v_{i,j}$ with $1 \leq i \leq N$ and $1 \leq j \leq |\overline{x}|$ is true, then the state variable $x_j \in \overline{x}$ appears as input of gate i, otherwise not.
- If parameter $k^n_{i,j}$ with $1 \leq i \leq N$ and $1 \leq j \leq |\overline{x}|$ is true, then gate i can only use the negated variable x_j as input, otherwise only the unnegated variable.
- If parameter $k^u_{i,j}$ with $1 \leq i \leq N$ and $1 \leq j < i$ is true, then the output of gate j appears as input of gate i, otherwise not.
- If parameter $k^m_{i,j}$ with $1 \leq i \leq N$ and $1 \leq j < i$ is true, then gate i can only use the negated output of gate j as input, otherwise only the unnegated output.
- The single parameter k^n defines if the output of the final gate defines $H(\overline{x}, \overline{k})$ or $\neg H(\overline{x}, \overline{k})$.

This gives $|\overline{k}| = N \cdot (2 \cdot |\overline{x}| + N - 1) + 1$ template parameters.

Example 21. We continue Example 20, where $\overline{x} = (x_1, x_2, x_3)$, $N = 3$ and $F(\overline{x}) = (x_1 \vee \neg x_2) \wedge (\neg x_3)$, which can be rewritten to $\neg(\neg x_1 \wedge x_2) \wedge (\neg x_3)$. This formula can be realized with

- $k^v_{2,1} = k^v_{2,2} = \mathsf{true}$ and $k^v_{2,3} = \mathsf{false}$ (gate 2 uses x_1 and x_2 as input but not x_3),
- $k^n_{2,1} = \mathsf{true}$ and $k^n_{2,2} = \mathsf{false}$ (gate 2 uses x_1 negated and x_2 unnegated),

- $k^u_{2,1} = \mathsf{false}$ (gate 2 ignores the output of gate 1),
- $k^v_{3,3} = \mathsf{true}$ and $k^v_{3,1} = k^v_{3,2} = \mathsf{false}$ (gate 3 uses x_3 as input but not x_1 and not x_2),
- $k^n_{3,3} = \mathsf{true}$ (gate 3 uses x_3 negated),
- $k^u_{3,2} = k^m_{3,2} = \mathsf{true}$ and $k^u_{3,1} = \mathsf{false}$ (gate 3 uses the negated output of gate 2, but ignores the output of gate 1), and
- $k^n = \mathsf{false}$ (the output $H(\overline{x}, \overline{k})$ is defined by the unnegated output of gate 3).

All other parameters are irrelevant. In particular, the output of gate 1 is completely ignored.

In our implementation, choosing N works in the same way as for the CNF template: starting with $N = 1$, N is increased by 1 in case of unsatisfiability of Equation 3.3 until $N = 4$ is reached. From there, N is doubled upon failure. There is a straightforward way to represent a CNF with N clauses as a network of $N + 1$ AND-gates. Hence, the criterion for detecting unrealizability with CNF templates can also be applied here: If Equation 3.3 is unsatisfiable for $N > 2^{|\overline{x}|}$, the specification must be unrealizable.

3.1.5.3 Implementation with SAT Solvers

In this section, we present an extension of the Counterexample-Guided Inductive Synthesis (CEGIS) approach that allows us to compute satisfying assignments of Equation 3.3 with SAT solvers instead of a QBF solver.

Basic idea. Recall that CEGIS (see Section 2.7) is an approach to compute satisfying assignments in formulas of the form $\exists \overline{e} : \forall \overline{u} : F(\overline{e}, \overline{u})$ by iterative refinements of a solution candidate. With

$$M(\overline{k}, \overline{x}, \overline{i}, \overline{c}, \overline{x}') = \big(I(\overline{x}) \rightarrow H(\overline{x}, \overline{k})\big) \wedge \big(H(\overline{x}, \overline{k}) \rightarrow P(\overline{x})\big) \wedge \big(H(\overline{x}, \overline{k}) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge H(\overline{x}', \overline{k})\big)\big)$$

being an abbreviation for the matrix of the QBF in Equation 3.3, our task is now to compute a satisfying assignment for the parameters $\overline{k}$ in $\exists \overline{k} : \forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : M(\overline{k}, \overline{x}, \overline{i}, \overline{c}, \overline{x}')$. Hence, there is an additional existential quantifier on the innermost level. This existential quantifier does not affect the computation of solution candidates significantly: Candidates are satisfying assignments for the variables $\overline{k}$ in

$$\bigwedge_{(\mathbf{x}, \mathbf{i}) \in D} \exists \overline{c}, \overline{x}' : M(\overline{k}, \mathbf{x}, \mathbf{i}, \overline{c}, \overline{x}'),$$

where the existential quantification of $\overline{c}$ and $\overline{x}'$ can be handled by renaming these variables in every copy of M and then calling a SAT solver. The computation of counterexamples, i.e., values for the variables $\overline{x}$ and $\overline{i}$, becomes more intricate, though. Instead of a satisfying assignment for $\neg F(\mathbf{e}, \overline{u})$, we now need to compute an assignment $\mathbf{x}, \mathbf{i}$ for the variables $\overline{x}, \overline{i}$ in

$$\neg \exists \overline{c}, \overline{x}' : M(\mathbf{k}, \overline{x}, \overline{i}, \overline{c}, \overline{x}'),$$

where $\mathbf{k}$ represents fixed values for the variables $\overline{k}$. The negation turns the existential quantification into a universal one. The resulting quantifier alternation prevents us from computing counterexamples with a single call to a SAT solver. A QBF solver could be used, but the idea of this section is to substitute QBF solving with plain SAT solving. Hence, we will use an iterative approach that is similar to SATWIN1 in Algorithm 3.4 to compute counterexamples.

Algorithm. The procedure TEMPLWINSAT in Algorithm 3.5 presents our solution. It takes as input a template $H(\overline{x}, \overline{k})$ for a winning area as well as a safety specification $\mathcal{S}$. As output, it returns either a concrete winning area $F(\overline{x})$ as an instantiation of the template $H(\overline{x}, \overline{k})$, or "$\mathsf{fail}$" of no instantiation of $H(\overline{x}, \overline{k})$ can be a winning area. The structure of the algorithm is the same as for CEGISSMT in Algorithm 2.5: The formula $G(\overline{k}, \overline{t})$ accumulates constraints that the template parameters $\overline{k}$ have to satisfy, where $\overline{t}$ is a vector of auxiliary variables. Line 4 computes candidate template parameter values $\mathbf{k}$ in form of a satisfying assignment for G. If the formula is unsatisfiable, then no template instantiation can be a winning area and the procedure returns "fail". If the formula is satisfiable, a candidate winning

Algorithm 3.5 TEMPLWINSAT: An algorithm to compute template instantiations using SAT solvers.

```
1: procedure TEMPLWINSAT$\big(H(\overline{x},\overline{k}), (\overline{x},\overline{i},\overline{c},I,T,P)\big)$, returns: A winning area $F(\overline{x})$ or "fail"
2:     $G(\overline{k},\overline{t}) := \mathsf{true}$
3:     while true do
4:         $(\mathsf{sat},\mathbf{k}) :=$ PROPSATMODEL$\big(G(\overline{k},\overline{t})\big)$
5:         if $\mathsf{sat} = \mathsf{false}$ then
6:             return "fail"
7:         end if
8:         $(\mathsf{correct},\mathbf{x},\mathbf{i}) :=$ CHECK$\big(H(\overline{x},\mathbf{k}), (\overline{x},\overline{i},\overline{c},I,T,P)\big)$
9:         if $\mathsf{correct} = \mathsf{true}$ then
10:            return $H(\overline{x},\mathbf{k})$
11:        end if
12:        $\overline{t}_c := \mathsf{CreateFreshCopy}(\overline{c})$
13:        $\overline{t}_x := \mathsf{CreateFreshCopy}(\overline{x}')$
14:        $G(\overline{k},\overline{t}) \;:=\; G(\overline{k},\overline{t}) \wedge \big(I(\mathbf{x}) \;\rightarrow\; H(\mathbf{x},\overline{k})\big) \wedge$
                                  $\big(H(\mathbf{x},\overline{k}) \;\rightarrow\; P(\mathbf{x})\big) \wedge$
                                  $\big(H(\mathbf{x},\overline{k}) \rightarrow \big(T(\mathbf{x},\mathbf{i},\overline{t}_c,\overline{t}_x) \wedge H(\overline{t}_x,\overline{k})\big)\big)$
15:    end while
16: end procedure
17: procedure CHECK$\big(F(\overline{x}), (\overline{x},\overline{i},\overline{c},I,T,P)\big)$, returns: $(\mathsf{correct},\mathbf{x},\mathbf{i})$
18:    $(\mathsf{sat},\mathbf{x}) :=$ PROPSATMODEL$\big((I(\overline{x}) \wedge \neg F(\overline{x})) \vee (F(\overline{x}) \wedge \neg P(\overline{x}))\big)$
19:    if sat then
20:        return $(\mathsf{true},\mathbf{x},\bigwedge_{i\in\overline{i}} \neg i)$
21:    end if
22:    $U(\overline{x},\overline{i}) := \mathsf{true}$
23:    while true do
24:        $(\mathsf{sat},\mathbf{x},\mathbf{i}) :=$ PROPSATMODEL$\big(F(\overline{x}) \wedge U(\overline{x},\overline{i}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg F(\overline{x}')\big)$
25:        if $\neg\mathsf{sat}$ then
26:            return $(\mathsf{true},\mathsf{true},\mathsf{true})$
27:        end if
28:        $(\mathsf{sat},\mathbf{c}) :=$ PROPSATMODEL$\big(F(\overline{x}) \wedge \mathbf{x} \wedge \mathbf{i} \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge F(\overline{x}')\big)$
29:        if $\neg\mathsf{sat}$ then
30:            return $(\mathsf{false},\mathbf{x},\mathbf{i})$
31:        else
32:            $U := U \wedge \neg$PROPMINUNSATCORE$\big(\mathbf{x}\wedge\mathbf{i}, \mathbf{c}\wedge F(\overline{x}) \wedge U(\overline{x},\overline{i}) \wedge T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \neg F(\overline{x}')\big)$
33:        end if
34:    end while
35: end procedure
```

area $F(\overline{x}) = H(\overline{x},\mathbf{k})$ is computed using the parameter values $\mathbf{k}$. Next, the candidate is checked in Line 8. This step is different to CEGISSMT in Algorithm 2.5 and explained in the next paragraph. If the candidate is correct, it is returned. Otherwise, the procedure CHECK returns a counterexample in form of a satisfying assignment $\mathbf{x},\mathbf{i}$ for the variables $\overline{x},\overline{i}$. The meaning of this counterexample is that

$$\exists\overline{c},\overline{x}' : \big(I(\mathbf{x}) \rightarrow H(\mathbf{x},\mathbf{k})\big) \wedge \big(H(\mathbf{x},\mathbf{k}) \rightarrow P(\mathbf{x})\big) \wedge \big(H(\mathbf{x},\mathbf{k}) \rightarrow \big(T(\mathbf{x},\mathbf{i},\overline{c},\overline{x}') \wedge H(\mathbf{x}',\mathbf{k})\big)\big)$$

does *not* hold, thus witnessing that $\mathbf{k}$ cannot be a solution to Equation 3.3 yet. To make sure the candidate of the next iteration works also for the counterexample $\mathbf{x},\mathbf{i}$, the constraints on $\overline{k}$ are refined accordingly in Line 14. The variables $\overline{c}$ and $\overline{x}'$ are renamed to fresh auxiliary variables in order to account for their existential quantification.

Counterexample computation. The procedure CHECK in Algorithm 3.5 is a helper routine for TEMPLWINSAT that checks if a given candidate $F(\overline{x})$ is a winning area. It returns $\mathsf{correct} = \mathsf{true}$ if this is the case. Otherwise, it sets $\mathsf{correct} = \mathsf{false}$ and returns a counterexample $\mathbf{x}, \mathbf{i}$ witnessing the incorrectness. Line 18 checks if the first two properties in the definition of a winning area F, namely $I \rightarrow F$ and $F \rightarrow P$, are satisfied (see Definition 11). If this is not the case, a satisfying assignment $\mathbf{x}$ is returned as a counterexample witnessing this defect. The input vector $\mathbf{i}$ returned as part of the counterexample is irrelevant in this case. Otherwise, CHECK turns to verifying the third property of a winning area, namely $F \rightarrow \mathsf{Force}_1^s(F)$. Here, we search for a counterexample $\mathbf{x}, \mathbf{i}$ such that no value $\mathbf{c}$ can prevent the system from leaving $F(\overline{x})$ if the environment picks input $\mathbf{i}$ from state $\mathbf{x} \models F(\overline{x})$. The same kind of counterexample computation was performed already by SATWIN1 in Algorithm 3.4, so we simply reuse this algorithm here. The difference is that $F(\overline{x})$ is not refined by CHECK. Thus, there is no need for lazy updates of $\neg F(\overline{x}')$, which renders quite some lines of Algorithm 3.4 obsolete.

An optimization. The check in Line 18 of Algorithm 3.5 can actually be omitted if we ensure that $\forall \overline{x}, \overline{k} : I(\overline{x}) \rightarrow H(\overline{x}, \overline{k})$ and $\forall \overline{x}, \overline{k} : H(\overline{x}, \overline{k}) \rightarrow P(\overline{x})$ holds by the construction of the template $H(\overline{x}, \overline{k})$. This can easily be achieved by taking any template $H'(\overline{x}, \overline{k})$ and defining a new template $H(\overline{x}, \overline{k}) = \big(H'(\overline{x}, \overline{k}) \wedge P(\overline{x})\big) \vee I(\overline{x})$, given that $I(\overline{x}) \wedge \neg P(\overline{x})$ is unsatisfiable (otherwise the specification is trivially unrealizable). We use this optimization in our implementation.

Incremental solving. Algorithm 3.5 is well suited for incremental SAT solving. We propose to use three solver instances. The first one stores G and is used for Line 4. Constraints are only added to G in Line 14, so no re-initialization is needed. The second solver instances stores $F(\overline{x}) \wedge U(\overline{x}, \overline{i}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \neg F(\overline{x}')$ and is used in Line 24 and 32. It is (re-)initialized when CHECK is called. After that, clauses are only added to U in Line 32. Finally, the third solver instance stores $F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')$ and is used in Line 28. This instance is also (re-)initialized whenever CHECK is called. This CNF does not change at all during the execution of CHECK. The conjunctions with $\mathbf{x}$, $\mathbf{i}$ and $\mathbf{c}$ are realized with assumption literals that are temporarily asserted.

3.1.5.4 Discussion

The template-based approach has a potential for finding simple winning areas quickly. There may exist many winning areas that satisfy the constraints given by Definition 11. The algorithms SAFEWIN, QBFWIN and SATWIN1 discussed earlier will always compute the largest possible winning area (modulo unreachable states if used with optimization RG or RC). The template-based approach is more flexible in this respect. As an extreme example, suppose that there is only one initial state, it is safe, and the system can enforce that the play stays in this state. Suppose further that the winning region is complicated. The template-based approach may find $F = I$ quickly, while the other approaches may require many iterations to compute the winning region.

On the other hand, the template-based approach can be expected to scale poorly if no simple winning area exists or if the synthesis problem is unrealizable. Starting with a small expressiveness parameter N, Equation 3.3 will be unsatisfiable, so N is increased. With increasing N, the search space for the solver increases, which results in longer execution times. For unrealizable specifications, we can only terminate once $N > 2^{|\overline{x}|}$ (when using our CNF or AND-inverter graph templates). Except for specifications with a very low numbers of state variables, a timeout is likely to be hit before this point can be reached.

3.1.6 Reduction to Effectively Propositional Logic (EPR)

The template-based approach presented in the previous section may work well if a simple representation of a winning area exists. However, one drawback is the need to select a template, which is a delicate matter. It would be more desirable to directly compute a winning area as a Skolem function of a quantified formula. Unfortunately, the definition of a winning area (Definition 11) not only involves the winning area $F(\overline{x})$ itself, but also its next-state copy $F(\overline{x}')$. Hence, we have to compute two Skolem functions, and the two functions have to be functionally consistent. This problem cannot be formulated as a QBF formula with a linear quantifier prefix, but requires more expressive logics.

3.1.6.1 Using Henkin Quantifiers

One solution is to use so-called Henkin quantifiers [108], which are quantifiers that are only partially ordered. This partial order can be used to restrict variable dependencies. In particular, a winning area $F(\overline{x})$ can be computed as a Skolem function for the variable w in

$$\begin{matrix}\forall \overline{x} : \exists w : \forall \overline{i} : \exists \overline{c} : \\ \forall \overline{x}' : \exists w' :\end{matrix} \quad \big(I(\overline{x}) \rightarrow w\big) \wedge \big(w \rightarrow P(\overline{x})\big) \wedge \big(w \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \rightarrow w'\big) \wedge \big((\overline{x} = \overline{x}') \rightarrow (w = w')\big).$$

This formulation ensures that the Skolem function $F(\overline{x})$ for w can only depend on $\overline{x}$, and the Skolem function $G(\overline{x}')$ for w' can only depend on $\overline{x}'$. The last constraint enforces functional consistency between F and G, i.e., F and G are actually the same function but applied to different parameters. The logic of applying Henkin quantifiers to propositional formulas is called Dependency Quantified Boolean Formulas (DQBF) and was first described by Peterson and Reif [173]. The problem of deciding whether a DQBF formula is satisfiable is NEXPTIME complete [173]. In addition to this high complexity, only a few approaches and tools to solve DQBF formulas have recently been proposed [92, 93]. For this reason, we did not implement a DQBF-based solution but we rather use EPR, where mature solvers are available.

3.1.6.2 Using Effectively Propositional Logic (EPR)

Recall from Section 2.2.5 that EPR is the set of first-order logic formulas of the form $\exists \overline{x} : \forall \overline{y} : F(\overline{x}, \overline{y})$, where F is a quantifier-free formula in CNF that must not contain function symbols but can contain predicate symbols. These predicate symbols are implicitly quantified existentially. We seek a winning area $F(\overline{x})$ satisfying the three properties of Definition 11, which can be combined to

$$\exists F : \forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : \big(I(\overline{x}) \rightarrow F(\overline{x})\big) \wedge \big(F(\overline{x}) \rightarrow P(\overline{x})\big) \wedge \big(F(\overline{x}) \rightarrow \big(T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge F(\overline{x}')\big)\big).$$

In order to transform this constraint into EPR, we need to perform several steps, which are similar to those by Seidl et al. [190] when transforming QBF formulas into EPR.

Step 1. We replace all the Boolean variables $\overline{x}, \overline{i}, \overline{c}, \overline{x}'$ by corresponding first-order domain variables. Since the original variables can only take two different values, we introduce a unary predicate V to represent the truth value of a domain variable. We also introduce two domain constants $\top$ and $\bot$ to encode true and false, and add the axioms $V(\top)$ and $\neg V(\bot)$ to the final EPR formula.

Step 2. We introduce predicate symbols $I(\overline{x})$, $P(\overline{x})$, $T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ and $F(\overline{x})$ to represent the different parts of the formula. The predicates I, P and T are equipped with additional constraints that fully define their truth value based on the truth values of the variables on which they depend. The predicate F is left unconstrained because it represents the winning area we wish to compute.

Step 3. The third step is to eliminate the existential quantification over $\overline{c}$ and $\overline{x}'$. Since the transition relation is both deterministic and complete (see Definition 5), the one-point rule (Equation 2.2) can be used to eliminate the existential quantification over $\overline{x}'$ by rewriting the formula to

$$\exists F : \forall \overline{x}, \overline{i} : \exists \overline{c} : \forall \overline{x}' : \big(I(\overline{x}) \rightarrow F(\overline{x})\big) \wedge \big(F(\overline{x}) \rightarrow P(\overline{x})\big) \wedge \big(\big(F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{c}, \overline{x}')\big) \rightarrow F(\overline{x}')\big).$$

The existential quantification over $\overline{c}$ is eliminated by Skolemization: for every variable $c_j \in \overline{c}$, we introduce a new predicate $C_j(\overline{x}, \overline{i})$. All occurrences of $V(c_j)$ in the definition of T are then replaced by $C_j(\overline{x}, \overline{i})$. This gives a formula of the form

$$\exists F, C_1, \ldots, C_{|\overline{c}|} : \forall \overline{x}, \overline{i}, \overline{x}' : \big(I(\overline{x}) \rightarrow F(\overline{x})\big) \wedge \big(F(\overline{x}) \rightarrow P(\overline{x})\big) \wedge \big(\big(F(\overline{x}) \wedge T(\overline{x}, \overline{i}, \overline{x}')\big) \rightarrow F(\overline{x}')\big).$$

Step 4. The body of the resulting formula needs to be encoded into CNF. Since we have a conjunction of implications on the top-level, this is mainly a matter of encoding the constraints defining I, P and

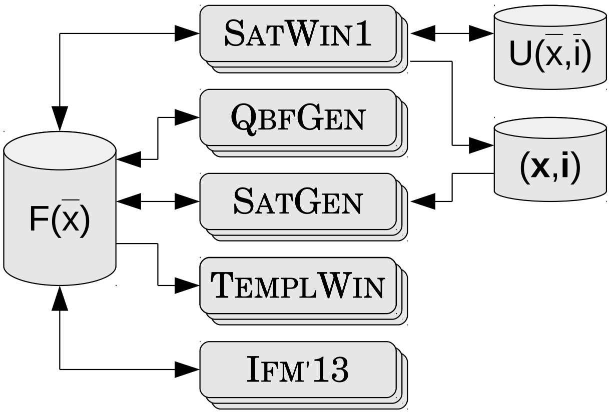


Figure 3.10: Overview of our parallelized strategy computation. Rectangles are threads running some methods. Arrows denote some flow of information.

T into CNF. Note that the standard Tseitin [208] or Plaisted-Greenbaum [174] transformations introduce new auxiliary variables that are quantified existentially on the innermost level. Since this is not allowed in EPR, these auxiliary variables need to be eliminated again. Similar to the elimination of the variables $\overline{c}$ in Step 3, we do this by introducing new predicates. To increase efficiency, we do not pass all variables of $\overline{x}, \overline{i}, \overline{x}'$ as arguments to the new predicates, but rather analyze the variable dependencies structurally and pass only the relevant ones.

Solving the resulting EPR formula. We call iProver on the resulting EPR formula. iProver is an instantiation-based first-order theorem prover that can also produce implementations for the predicates that occur in the formula. This means that the solver directly returns a winning area $F(\overline{x})$. Moreover, since we represent the truth values of the variables $c_j \in \overline{c}$ with predicates $C_j(\overline{x}, \overline{i})$, we can also directly extract an implementation from the solver result.[4] That is, there is no need to apply the circuit construction methods that will be presented in Section 3.2 when using the EPR synthesis approach.

Discussion. Similar to the template-based approach presented in Section 3.1.5, this approach does not compute the winning region but some winning area. It can thus benefit from situations where the winning region is complicated but a simple winning area exists. In contrast to the template-based approach, it eliminates the need to guess a template and to increase the expressiveness of the template if no solution is found. The price that is payed for this benefit is the higher worst-case complexity for checking the satisfiability of the constructed formulas because a more expressive logic is used.

3.1.7 Parallelization

The various methods for strategy computation presented so far have different strengths and weaknesses and, consequently, perform well on different classes of benchmarks. To a smaller extent, different characteristics can also be observed within one method when run with different optimizations or solvers. In this section we therefore combine different methods and configurations in the hope to inherit all their strengths while compensating their weaknesses. We do this in a parallelized way, where individual methods are running in separate threads but share discovered information that may be helpful for others.

Figure 3.10 gives a proposal for combining a promising subset of the methods (or fragments thereof). Arrows denote information that is exchanged between threads.

SatWin1 threads. The SatWin1 threads execute the SatWin1 procedure from Algorithm 3.4 and can be seen as the main workhorse. Individual SatWin1 threads can be run with or without optimization RG, with or without quantifier expansion, and with different SAT solvers. All newly discovered clauses of the winning region $F(\overline{x})$ are put into a central database and communicated to the other threads. Newly

[4] Because of the poor scalability of the EPR approach in our experiments, we did not implement a parser for the predicate implementations returned by iProver in our tool yet.

discovered U-clauses are also shared between SATWIN1 threads. In order for this to work, the SATWIN1 threads need be synchronized regarding their restarts of solverC, i.e., they need to work with the same version of $\neg G(\overline{x}')$ at any time. If several SATWIN1 threads are running in a mode where they perform universal expansion, the expansion is only done by one thread (while the others sleep) in order not to waste resources (like stressing the memory bus unnecessarily).

QBFGEN threads. The QBFGEN threads take existing clauses from F and attempt to generalize them further by eliminating more literals. This is done as in Line 9 to Line 15 of the QBFWIN procedure in Algorithm 3.1 using a QBF solver. If a clause could be shortened, the reduced clause is communicated to all other threads. Individual QBFGEN threads can be run with or without optimization RG, with or without QBF preprocessing, and with or without incremental QBF solving (the combination of incremental solving plus preprocessing is not available).

SATGEN threads. These threads take counterexamples in form of state-input pairs $(\mathbf{x}, \mathbf{i})$ as computed by the SATWIN1 threads and compute *all* generalizations using a SAT solver (as illustrated in Figure 3.2). The resulting F-clauses are communicated to all other threads.

TEMPLWIN threads. These threads implement the template-based method from Section 3.1.5, using CNF templates of increasing size. The clauses from F are considered as fixed over-approximation of the winning area to compute — the threads only compute additional clauses such that a winning area is obtained. A timeout of 20 seconds makes the thread try again (with a potentially refined set F of fixed clauses) if a solution cannot be found quickly. The short timeout is justified by the observation that the template-based approach either finds a solution quickly or not at all. The QBF-based implementation and the SAT-based implementation of the template-based approach are alternated from timeout to timeout. The TEMPLWIN threads are information sinks: the only information communicated back to other threads is a request to terminate if a solution has been found.

IFM'13 threads. These threads execute a reimplementation of the SAT-based synthesis method proposed by Morgenstern et al. [163]. This method maintains an over-approximation $G(\overline{x})$ of the winning region $W(\overline{x})$ as well as over-approximations of sets of states from which the environment can win the game in different numbers of steps. We couple $G(\overline{x})$ with $F(\overline{x})$: If new clauses are added to $G(\overline{x})$, then they are also added to $F(\overline{x})$ and communicated to the other threads. If other threads discover new F-clauses, they are also added to G in the IFM'13 threads.

Configuration. When only one thread is available, we make it execute SATWIN1 with optimization RG, quantifier expansion and MiniSat as underlying SAT solver. If two threads are available, the second one executes TEMPLWIN (with DepQBF, Bloqqer and MiniSat). If three threads are available, the third thread runs IFM'13 using MiniSat. With four threads, we also use a second instance of SATWIN1, but with quantifier expansion disabled. With five threads, we also include a SATGEN thread, and with six threads we also include a QBFGEN thread.

Variations. The current realization always shares *all* discovered clauses that refine the winning region with all other threads. Another option is to share only *small* clauses (where the number of literals is below some threshold) in order to reduce the communication overhead. In general, smaller clauses refine the winning region more substantially than larger ones, so this approach would focus on communicating only significant findings. Another promising extension is to include also threads that run BDD-based algorithms, e.g., a BDD-based realization of Algorithm 2.1. The BDD-based threads can directly use clauses discovered by other threads to refine the BDD that represents the winning region. Communication in the other direction is possible as well: many BDD libraries provide functions to convert a BDD into CNF. While it may be expensive to share all clauses of such a CNF translation, it may still be beneficial to factor out a set of small clauses and communicate them.

Discussion. The main purpose of our parallelization is to combine different methods that complement each other. Exploiting hardware parallelism in only a secondary aspect because, due to the high worst-case complexities, even a speedup factor of, say, 10 may have little impact on the ability of solving larger benchmark instances. Furthermore, we do not claim that our choice of distributing workload over the threads is in any way optimal. We rather selected the methods to run in individual threads quite greedily, based on the performance results when running methods in isolation (see Section 3.3) and based on experiments with subsets of the benchmarks. However, there is such a plethora of possibilities for combining different methods, fragments thereof, optimizations, heuristics and solver configurations that finding particularly good configurations is quite an intricate task. Hence, we rather see the main contribution of our parallelization in providing a "playground" for combining different approaches and configurations. It demonstrates that a parallelized way of combining different SAT-based synthesis approaches is easily possible. This stands in contrast to BDD-based synthesis algorithms, where a parallelization is often much more difficult to achieve. Our parallelization goes far beyond a pure portfolio approach because fine-grained information about refinements of the winning region, discovered counterexamples and unsuccessful attempts to compute counterexamples is exchanged between the threads as soon as discovered. This information can speed up the progress in other threads and thus stimulate "cross-fertilization" effects.

3.2 From Strategies to Circuits

In Section 3.1, we presented a number of SAT-based methods to compute a strategy for defining the control signals $\overline{c}$ always in such a way that a given safety specification is enforced. Recall that such a strategy is a formula $S(\overline{x},\overline{i},\overline{c},\overline{x}')$ such that

$$\forall \overline{x},\overline{i} : \exists \overline{c},\overline{x}' : S(\overline{x},\overline{i},\overline{c},\overline{x}').$$

That is, for every state $\mathbf{x}$ and input $\mathbf{i}$, the strategy will contain at least one vector of control values $\mathbf{c}$ that is allowed in this situation. In many situations, many control values can be allowed, though. The task is now to compute a system implementation in form of a function $f : 2^{\overline{x}} \times 2^{\overline{i}} \rightarrow 2^{\overline{c}}$ to uniquely define the control signals $\overline{c}$ based on the current state variables $\overline{x}$ and the uncontrollable inputs $\overline{i}$. The system implementation f is supposed to implement the strategy in the sense that

$$\forall \overline{x},\overline{i} : \exists \overline{x}' : S\big(\overline{x},\overline{i}, f(\overline{x},\overline{i}),\overline{x}'\big)$$

holds. That is, for all concrete assignments $\mathbf{x},\mathbf{i}$, the control variable assignment $\mathbf{c} = f(\mathbf{x},\mathbf{i})$ computed by f must be allowed by the strategy S. Finally, this function f needs to be implemented as a circuit. Obviously, we prefer fast algorithms that produce small circuits. In order to achieve this, the freedom in the strategy relation S needs to be exploited cleverly.

A cofactor-based algorithm to solve the problem has already been presented in Section 2.5.3. It can be seen as the "standard method" for computing an implementation from a strategy, and can easily be implemented using BDDs. In the following subsections, we will present alternative approaches that use SAT- or QBF solvers instead. The presented approaches are not specific to safety specifications. However, in many cases, the specific structure

$$S(\overline{x},\overline{i},\overline{c},\overline{x}') = T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(W(\overline{x}) \rightarrow W(\overline{x}')\big)$$

of strategies for safety specifications can be exploited. We will thus always present the general approach first, and then discuss an efficient implementation for safety synthesis problems. As a preprocessing step to all our methods, we simplify W by calling CompressCnf (see Algorithm 3.2) with literal dropping enabled in order to remove redundant literals and clauses from W. As a postprocessing step to all our methods, we invoke the tool ABC [42] in order to reduce the size of the produced circuits. These steps will not be mentioned explicitly in the following subsections.

3.2.1 QBF Certification

A system implementation can be computed in form of a Skolem function for the signals $\overline{c}$ in the QBF $\forall \overline{x},\overline{i} : \exists \overline{c},\overline{x}' : S(\overline{x},\overline{i},\overline{o},\overline{x}')$. The QBFCert [168] framework by Niemetz et al. computes such Skolem

Algorithm 3.6 NEGLEARN: Computing a CNF representation for the negation of a formula $F(\overline{x})$.

```
procedure NEGLEARN($F(\overline{x})$), returns: $\neg F(\overline{x})$ in CNF
    $N(\overline{x}) :=$ true
    while sat = true in $(\mathsf{sat}, \mathbf{x}) :=$ PROPSATMODEL$\big(F(\overline{x}) \wedge N(\overline{x})\big)$ do
        $N(\overline{x}) := N(\overline{x}) \wedge \neg$PROPMINUNSATCORE$\big(\mathbf{x}, \neg F(\overline{x})\big)$
    end while
    return $N(\overline{x})$
end procedure
```

functions for satisfiable QBFs from proof traces produced by the DepQBF [155] solver. The resulting Skolem functions are produced as circuits in AIGER format. Hence, in our setting, a single call to QBFCert suffices to compute a system implementation in form of a circuit.

3.2.1.1 Efficient Implementation for Safety Synthesis Problems

While the basic approach is simple, we can still apply some optimizations to increase the efficiency for the case of safety synthesis problems.

QBF formulation. Instead of computing a Skolem function for the variables $\overline{c}$ in the formula

$$\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(W(\overline{x}) \rightarrow W(\overline{x}')\big) \tag{3.4}$$

we rather compute a Herbrand function in its negation

$$\exists \overline{x}, \overline{i} : \forall \overline{c}, \overline{x}' : \neg T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \vee \big(W(\overline{x}) \wedge \neg W(\overline{x}')\big).$$

Because T is both deterministic and complete (Definition 5), the one-point rule (Equation 2.2) can be applied to turn the universal quantification over $\overline{x}'$ into an existential quantification:

$$\exists \overline{x}, \overline{i} : \forall \overline{c} : \exists \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge W(\overline{x}) \wedge \neg W(\overline{x}'). \tag{3.5}$$

Just like most QBF solvers, QBFCert requires a PCNF as input. Since most of our methods to compute a winning region (or winning area) produce W in CNF, we only need to transform T and $\neg W(\overline{x}')$ into CNF. In contrast, using Equation 3.4 would require an additional CNF encoding of the implication $W(\overline{x}) \rightarrow W(\overline{x}')$. Another advantage of using Equation 3.5 lies in the size of the proofs: since the QBF is now unsatisfiable, the QBFCert framework processes a clause resolution proof instead of a cube resolution proof. These clause resolution proofs are often smaller.

Negation of $W(\overline{x}')$. For complex benchmarks, the auxiliary files produced by QBFCert can still grow very large (hundreds of GB). One reason is that a straightforward CNF encoding of $\neg W(\overline{x}')$ requires many auxiliary variables and clauses. We can reduce the size of the auxiliary files (by up to a factor of 30 in our experiments) by computing a CNF representation of $\neg W(\overline{x}')$ without introducing auxiliary variables. The procedure NEGLEARN in Algorithm 3.6 computes such a negation with query learning. It follows the principle of CNFLEARN, shown in Algorithm 2.4, and uses a SAT solver to implement the queries: As long as N is not yet equivalent to $\neg F$, i.e., $F \wedge N$ is still satisfiable, NEGLEARN refines N with a clause that excludes the cube $\mathbf{x}$ witnessing this insufficiency. By taking the unsatisfiable core, the clause eliminates also other counterexamples. Since clauses are only added to N, NEGLEARN is well suited for incremental SAT solving.

3.2.1.2 Discussion

Dependencies between control signals. In contrast to COFSYNT from Algorithm 2.2, the QBF certification approach computes a circuit for all control signals simultaneously. This can be both an advantage and a disadvantage. The advantage is that dependencies between control signals can potentially be handled more effectively. COFSYNT can only take local decisions and fixes an implementation for one control signal without considering the consequences on other control signals (as long as some solution for the other signals still exist). The QBF certification approach is free to make global decisions when fixing the individual circuits. On the other hand, considering all control signals simultaneously instead of decomposing the problem into smaller subproblems can also be a scalability disadvantage.

Dependencies on reasoning engine. The performance of QBFCert as well as the quality of the resulting circuit depend on the ability of DepQBF to find a compact unsatisfiability proof quickly. In this sense, the technique is strongly dependent on the underlying symbolic reasoning engine. This is similar to COFSYNT when implemented using BDDs, where the ability to find a good variable ordering can influence the circuit size and the execution time heavily.

3.2.2 QBF-Based Query Learning

In this section, we introduce an approach that is also based on QBF solving, but constructs circuits for one control signal after the other. In this respect, it is more similar to COFSYNT presented in Algorithm 2.2. However, in contrast to COFSYNT, we will rely on query learning to exploit the implementation freedom in the strategy in order to obtain small circuits.

The query learning algorithms introduced in Section 2.6 compute a certain representation of a given target formula $G(\overline{x})$ precisely. That is, the resulting formula $F(\overline{x})$ will be equivalent to the target $G(\overline{x})$. This is achieved by starting with some initial approximation for F, and refining this approximation based on counterexamples witnessing that $F \neq G$. These counterexamples are also generalized to speed up the progress. The same formula G is used both for computing counterexamples and for generalizing them. However, by using two different formulas G_1 and G_2 in these two phases, we can also compute a function F such that $G_1 \rightarrow F \rightarrow G_2$. This idea can be used to exploit freedom in defining F, where the freedom is defined by (the difference between) G_1 and G_2. Note that F is actually an interpolant for $G_1 \wedge \neg G_2$ (see Section 2.2.1). Thus, this way of query learning with freedom can be seen as a special way to compute interpolants. However, depending on the underlying reasoning engine used in query learning, the formulas G_1 and G_2 do not have to be quantifier-free. Furthermore, by choosing an appropriate learning algorithm, we can control the shape of F. For instance, a CNF learning algorithm will produce F in form of a CNF formula.

In the following, we will present a circuit synthesis algorithm based on CNF learning using a QBF solver. CNF learning is particularly suitable in this setting because QBF solvers require formulas in PCNF, so building up the solution in CNF reduces the overhead (especially in terms of formula size) imposed by CNF transformations. Solutions with other learning algorithms can be found in our FMCAD'12 [82] publication. After introducing the basic algorithm, we will also discuss an efficient realization for safety synthesis problems.

3.2.2.1 QBF-Based CNF Learning

The procedure QBFSYNT in Algorithm 3.7 presents a CNF learning algorithm, implemented using a QBF solver. It synthesizes a circuit from a given strategy $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ while exploiting the freedom in S in order to obtain small circuits. QBFSYNT does not return any result but directly dumps the produced circuits. Individual circuits are computed for one $c_j \in \overline{c}$ after the other. In this respect, QBFSYNT is similar to COFSYNT (Algorithm 2.2) but different from QBF certification as presented in Section 3.2.1.

Definition of M_1 and M_0. Line 3 of QBFSYNT computes the formula $M_1(\overline{x}, \overline{i})$, which characterizes the set of all $(\overline{x}, \overline{i})$-assignments for which the current control signal c_j must be set to true: Recall from COFSYNT (Algorithm 2.2) that the formula

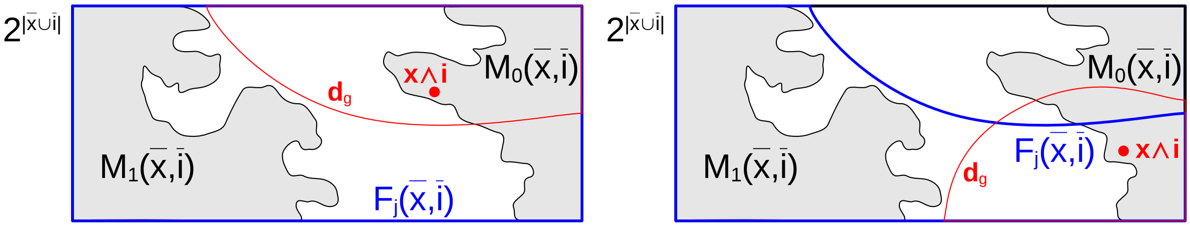
$$C_0(\overline{x}, \overline{i}) := \exists \overline{x}', \overline{c} : S\big(\overline{x}, \overline{i}, (c_0, \ldots, c_{j-1}, \mathsf{false}, c_{j+1}, \ldots, c_n), \overline{x}'\big)$$

Algorithm 3.7 QbfSynt: Synthesizing circuits from strategies with QBF-based CNF learning.

```
 1: procedure QbfSynt($S(\overline{x},\overline{i},\overline{c},\overline{x}')$)
 2:     for $c_j \in \overline{c}$ do
 3:         $M_1(\overline{x},\overline{i}) := \forall \overline{c},\overline{x}' : \neg S\big(\overline{x},\overline{i},(c_0,\dots,c_{j-1},\mathsf{false},c_{j+1},\dots,c_n),\overline{x}'\big)$
 4:         $M_0(\overline{x},\overline{i}) := \forall \overline{c},\overline{x}' : \neg S\big(\overline{x},\overline{i},(c_0,\dots,c_{j-1},\mathsf{true},c_{j+1},\dots,c_n),\overline{x}'\big)$
 5:         $F_j(\overline{x},\overline{i}) := \mathsf{true}$
 6:         while $\mathsf{sat}$ in $(\mathsf{sat},\mathbf{x},\mathbf{i}) :=$ QbfSatModel$\big(\exists \overline{x},\overline{i} : F_j(\overline{x},\overline{i}) \wedge M_0(\overline{x},\overline{i})\big)$ do
 7:             $\mathbf{d}_g :=$ QbfGeneralize$\big(\mathbf{x} \wedge \mathbf{i}, M_1(\overline{x},\overline{i})\big)$
 8:             $F_j(\overline{x},\overline{i}) := F_j(\overline{x},\overline{i}) \wedge \neg \mathbf{d}_g$
 9:         end while
10:         dumpCircuit$\big(c_j, F_j(\overline{x},\overline{i})\big)$
11:         $S(\overline{x},\overline{i},\overline{c},\overline{x}') := S(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(c_j \leftrightarrow F_j(\overline{x},\overline{i})\big)$
12:     end for
13: end procedure
14: procedure QbfGeneralize$\big(\mathbf{d}, M_1(\overline{x},\overline{i})\big)$, returns: $\mathbf{d}_g \subseteq \mathbf{d}$ such that $\mathbf{d}_g \wedge M_1$ is unsatisfiable
15:     $\mathbf{d}_g := \mathbf{d}$
16:     for each literal $l$ in $\mathbf{d}_g$ do
17:         $\mathbf{d}_t := \mathbf{d}_g \setminus \{l\}$
18:         if $\neg$QbfSatModel$\big(\exists \overline{x},\overline{i} : \mathbf{d}_t \wedge M_1(\overline{x},\overline{i})\big)$ then
19:             $\mathbf{d}_g := \mathbf{d}_t$
20:         end if
21:     end for
22:     return $\mathbf{d}_g$
23: end procedure
```

characterizes the set of all $(\overline{x},\overline{i})$-assignments for which $c_j = \mathsf{false}$ is allowed by the strategy S. Its negation $M_1(\overline{x},\overline{i}) = \neg C_0(\overline{x},\overline{i})$ is thus the set of all situations where $c_j = \mathsf{false}$ is not allowed by S, i.e., where c_j must be set to true. Analogously, the formula $M_0(\overline{x},\overline{i})$ represents the set of all $(\overline{x},\overline{i})$-assignments for which c_j must be false.

Learning an implementation F_j. The lines 5 to 9 compute a CNF formula $F_j(\overline{x},\overline{i})$ such that $M_1(\overline{x},\overline{i}) \rightarrow F_j(\overline{x},\overline{i}) \rightarrow \neg M_0(\overline{x},\overline{i})$ using a variant of the CnfLearn procedure from Algorithm 2.4. The first implication $M_1 \rightarrow F_j$ ensures that F_j is true whenever c_j must be true. The second implication $F_j \rightarrow \neg M_0$ ensures that whenever F_j is true, c_j does not have to be false. Together, these two conditions fully describe a proper implementation for c_j. Just like CnfLearn, we start with $F_j = \mathsf{true}$ (Line 5). Next, Line 6 checks if F_j is already correct in the sense that $M_1 \rightarrow F_j \rightarrow \neg M_0$ holds. The algorithm maintains the invariant $M_1 \rightarrow F_j$, so only $F_j \rightarrow \neg M_0$ needs to be checked. This is done by calling a QBF solver to search for a satisfying assignment $\mathbf{x},\mathbf{i} \models F_j \wedge M_0$ to the variables $\overline{x},\overline{i}$ for which F_j is true but c_j must be false. Note that M_0 contains a universal quantification of $\overline{c}$ and $\overline{x}'$, so a SAT solver cannot be used for this check. If no such counterexample $\mathbf{x},\mathbf{i}$ exists, the **while**-loop terminates. Otherwise, the counterexample cube $\mathbf{d} = \mathbf{x} \wedge \mathbf{i}$ is generalized into a cube $\mathbf{d}_g \subseteq \mathbf{d}$ by eliminating literals as long as $\mathbf{d}_g \wedge M_1$ is unsatisfiable. This is implemented in the subroutine QbfGeneralize and ensures that $\mathbf{d}_g$ does not contain any $(\overline{x},\overline{i})$-assignments for which c_j must be true, so it is safe to update F_j to $F_j \wedge \neg \mathbf{d}_g$ while preserving the invariant $M_1 \rightarrow F_j$. This update eliminates the original counterexample $\mathbf{d}$ for which F_j *must* be false. Due to the generalization, other $(\overline{x},\overline{i})$-assignments for which F_j *can* be false are also mapped to false. Going with "can be false" rather than "must be false" in the generalization phase results in potentially smaller clauses being added to F_j. This increases the potential for eliminating counterexamples before they are encountered in Line 6. Hence, exploiting the freedom between "must be false" and "can be false" — as done by QbfSynt — potentially does not only result in a more compact CNF representation of F_j but also in fewer iterations.


$2^{|\overline{x}\cup\overline{i}|}$
$\mathbf{x}\wedge\mathbf{i}$
$M_0(\overline{x},\overline{i})$
$\mathbf{d}_g$
$M_1(\overline{x},\overline{i})$
$F_j(\overline{x},\overline{i})$
$2^{|\overline{x}\cup\overline{i}|}$
$M_0(\overline{x},\overline{i})$
$F_j(\overline{x},\overline{i})$
$\mathbf{x}\wedge\mathbf{i}$
$M_1(\overline{x},\overline{i})$
$\mathbf{d}_g$

(a) First counterexample.

(b) Second counterexample.

Figure 3.11: Working principle of QBFWIN. The boxes represent the set of all $(\overline{x}, \overline{i})$-assignments. With $F_j = \mathsf{true}$, subfigure (a) illustrates the computation of a first counterexample $\mathbf{x}, \mathbf{i} \models F_j \wedge M_0$ as well as its generalization into a larger region $\mathbf{d}_g$, while $\mathbf{d}_g$ does not intersect with M_1. With $F_j = \neg\mathbf{d}_g$, subfigure (b) illustrates the computation of the second counterexample $\mathbf{x}, \mathbf{i} \models F_j \wedge M_0$ as well as its generalization.

Circuit construction and resubstitution. The remaining parts of QBFSYNT are the same as for COFSYNT (Algorithm 2.2): Line 10 dumps the formula $F_j(\overline{x}, \overline{i})$ in form of a circuit which defines c_j to be true whenever $F_j(\overline{x}, \overline{i})$ evaluates to true. This can easily be done by replacing every Boolean operator in F_j with the corresponding gate. We do not attempt to reuse existing gates while dumping the circuit, but leave this optimization to ABC [42] in the postprocessing step. Finally, Line 11 refines the strategy S with the solution for c_j to propagate consequences of fixing c_j on other control signals.

Auxiliary variables. If the strategy formula S contains auxiliary variables (e.g., from Tseitin-transformations [208]), then these variables are all handled as if they were part of $\overline{x}$. The resubstitution step in Line 11 may also introduce additional auxiliary variables, which are also handled like $\overline{x}$.

Illustration. Figure 3.11 illustrates the computation of a circuit for one control signal c_j graphically. The boxes represent the set $2^{|\overline{x}\cup\overline{i}|}$ of all possible assignments to the variables $\overline{x}$ and $\overline{i}$. Figure 3.11a depicts the initial situation. The region M_1 represents the set of all situations where c_j must be true, and M_0 represents the situations where c_j must be false. The definition of the strategy ensures that these two regions cannot overlap. The current approximation F_j of the solution is depicted in blue. Initially, $F_j = \mathsf{true}$ (Line 5 in QBFSYNT). Next, a counterexample $\mathbf{x}, \mathbf{i} \models F_j \wedge M_0$ is computed (Line 6). It is drawn as a red dot in Figure 3.11a. The counterexample cube $\mathbf{x} \wedge \mathbf{i}$ is then generalized into a larger region $\mathbf{d}_g$ by eliminating literals as long as $\mathbf{d}_g$ does not intersect with M_1. This is ensured by the check in Line 18 of QBFSYNT. Next, F_j is refined by subtracting the resulting region $\mathbf{d}_g$. The refined formula F_j is shown as a blue outline Figure 3.11b. Since the first counterexample is no longer contained in $F_j \wedge M_0$, it cannot be encountered again. Instead, the algorithm computes a different counterexample, which is generalized in the same way. This is illustrated in Figure 3.11b. After subtracting the second $\mathbf{d}_g$ from F_j (which is not shown in Figure 3.11), F_j does not intersect with M_0 any more. Hence there are no more situations where F_j is true but must be false. Since we did not remove any situation that is contained in M_1 from F_j, the final solution satisfies $M_1 \rightarrow F_j \rightarrow \neg M_0$ and the **while**-loop in QBFSYNT terminates. That is, F_j exploits the freedom between M_1 and M_0. Compared to learning a CNF formula for $\neg M_0$ precisely, this potentially reduces the number of iterations and the resulting circuit size, especially if $\neg M_0$ is complicated. In Figure 3.11, this is indicated by M_0 being more irregular in shape than F_j.

3.2.2.2 Efficient Implementation for Safety Synthesis Problems

The procedure SAFEQBFSYNT in Algorithm 3.8 presents an efficient realization of QBFSYNT for the case of safety specifications, where the winning strategy $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is defined via a winning region (or a winning area) $W(\overline{x})$. To make the QBF queries efficient, our aim is to avoid disjunctions and negations of subformulas as much as possible, and to reduce the amount of universal quantification.

Grouping of control variables. In every iteration, SAFEQBFSYNT splits the control variables $\overline{c}$ into three groups $\overline{c}_a$, c_j, $\overline{c}_b$: The single variable c_j is the one for which a circuit is constructed in the current

Algorithm 3.8 SafeQbfSynt: Synthesizes circuits from winning areas with QBF-based CNF learning.

```
 1: procedure SafeQbfSynt($T(\overline{x},\overline{i},\overline{c},\overline{x}')$, $W(\overline{x})$)
 2:     $T'(\overline{x},\overline{i},\overline{c},\overline{x}') := T(\overline{x},\overline{i},\overline{c},\overline{x}'),\quad \overline{c}_b := \overline{c},\quad \overline{c}_a := \emptyset$
 3:     for all $j$ from 1 to $|\overline{c}|$ do
 4:         $\overline{c}_b := \overline{c}_b \setminus \{c_j\}$
 5:         $M_1(\overline{x},\overline{i}) := \forall \overline{c}_b : \exists \overline{c}_a, \overline{x}' : T'\big(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_b,\overline{x}'\big) \wedge W(\overline{x}) \wedge \neg W(\overline{x}')$
 6:         $M_0(\overline{x},\overline{i}) := \forall \overline{c}_b : \exists \overline{c}_a, \overline{x}' : T'\big(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_b,\overline{x}'\big) \wedge W(\overline{x}) \wedge \neg W(\overline{x}')$
 7:         $F_j(\overline{x},\overline{i}) := \mathsf{true}$
 8:         while sat in $(\mathsf{sat},\mathbf{x},\mathbf{i}) :=$ QbfSatModel$\big(\exists \overline{x},\overline{i} : F_j(\overline{x},\overline{i}) \wedge M_0(\overline{x},\overline{i})\big)$ do
 9:             $\mathbf{d}_g :=$ QbfGeneralize$\big(\mathbf{x} \wedge \mathbf{i}, M_1(\overline{x},\overline{i})\big)$
10:             $F_j(\overline{x},\overline{i}) := F_j(\overline{x},\overline{i}) \wedge \neg \mathbf{d}_g$
11:         end while
12:         dumpCircuit$\big(c_j, F_j(\overline{x},\overline{i})\big)$
13:         $T'(\overline{x},\overline{i},\overline{c},\overline{x}') := T'(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(c_j \leftrightarrow F_j(\overline{x},\overline{i})\big)$
14:         $\overline{c}_a := \overline{c}_a \cup \{c_j\}$
15:     end for
16: end procedure
17: procedure QbfGeneralize$\big(\mathbf{d}, M_1(\overline{x},\overline{i})\big)$, returns: $\mathbf{d}_g \subseteq \mathbf{d}$ such that $\mathbf{d}_g \wedge M_1$ is unsatisfiable
18:     $\mathbf{d}_g := \mathbf{x} \wedge \mathbf{i}$
19:     for each literal $l$ in $\mathbf{d}$ do
20:         $\mathbf{d}_t := \mathbf{d}_g \setminus \{l\}$
21:         if $\neg$QbfSatModel$\big(\exists \overline{x},\overline{i} : \mathbf{d}_t \wedge M_1(\overline{x},\overline{i})\big)$ then
22:             $\mathbf{d}_g := \mathbf{d}_t$
23:         end if
24:     end for
25:     return $\mathbf{d}_g$
26: end procedure
```

iteration, $\overline{c}_a$ contains all variables for which a circuit has already been computed, and $\overline{c}_b$ contains all control variables for which a circuit will be computed in some future iteration. This split is performed in the Lines 2, 4 and 14, and will allow us to reduce the amount of universal quantification.

Definition of M_1 and M_0. With $S(\overline{x},\overline{i},\overline{c},\overline{x}') = T(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)$, we can apply the following transformations to compute a CNF for M_1 more efficiently.

$$\begin{aligned}
M_1(\overline{x},\overline{i}) =& \forall \overline{c}, \overline{x}' : \neg S\big(\overline{x},\overline{i},(c_0,\dots,c_{j-1},\mathsf{false},c_{j+1},\dots,c_n),\overline{x}'\big)\\
=& \forall \overline{c}_b, \overline{c}_a, \overline{x}' : \neg\Big(T(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_b,\overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)\Big)\\
=& \forall \overline{c}_b, \overline{c}_a, \overline{x}' : \Big(T(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_b,\overline{x}') \rightarrow \big(W(\overline{x}) \wedge \neg W(\overline{x}')\big)\Big)
\end{aligned}$$

SafeQbfSynt keeps a copy T' of the transition relation T. It is updated in such a way that all variables in $\overline{c}_a, \overline{x}'$ are defined uniquely by T'. For the variables $\overline{x}'$, this holds initially. For $\overline{c}_a$, this is ensured by Line 13. Thus, by using T' instead of T and applying the one-point rule (Equation 2.2), the universal quantification of $\overline{c}_a, \overline{x}'$ can be turned into an existential one:

$$M_1(\overline{x},\overline{i}) = \forall \overline{c}_b : \exists \overline{c}_a, \overline{x}' : \Big(T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_b,\overline{x}') \wedge W(\overline{x}) \wedge \neg W(\overline{x}')\Big)$$

The computation of $M_0(\overline{x},\overline{i})$ works analogously. As a result, only the control signals $\overline{c}_b$, for which no solution has been computed yet, are quantified universally in the QBF queries of Line 8 and 21. The variable vector $\overline{c}_b$ becomes shorter from iteration to iteration, which means that the formula gets "more propositional". In the last iteration, a SAT solver can actually be used instead of a QBF solver.

CNF conversion. The QBF queries in Line 8 and 21 contain only conjunctions. The formula F_j is always in CNF. Furthermore, most of our methods to compute a winning region or a winning area produce $W(\overline{x})$ in CNF. Hence, we only need to compute a CNF representation of T' and $\neg W(\overline{x}')$. The procedure NEGLEARN (Algorithm 3.6), which negates a formula without introducing auxiliary variables, was beneficial in the QBF certification approach but does not pay off in the learning-based approach. Hence, we apply the method of Plaisted and Greenbaum [174] to compute a CNF for $\neg W(\overline{x}')$.

QBF preprocessing. With our extension of Bloqqer [189] to preserve satisfying assignments, QBF preprocessing can be applied both for counterexample computation and generalization. However, while preprocessing was vital in our methods for computing a winning region, it does not give a significant speedup for SAFEQBFSYNT (see Section 3.3).

Incremental QBF solving. SAFEQBFSYNT is very well suited for incremental QBF solving, especially with a solver interface such as the one provided by DepQBF [156, 157]. We propose to use two solver instances incrementally. The first instance stores $F_j \wedge M_0$ and is used for Line 8. Since Line 10 only adds clauses to F_j, this solver instance is only re-initialized when a mayor iteration (synthesizing the next c_j) is started. The second solver instance stores M_1, is used for Line 21, and is also re-initialized when a mayor iteration starts. Before executing the loop in Line 19, we let the second solver instance compute an unsatisfiable core $\mathbf{d}_g$ of $\mathbf{d} = \mathbf{x} \wedge \mathbf{i}$ and only reduce this core further in the loop. The conjunction with $\mathbf{d}_t$ is realized with assumption literals.

3.2.2.3 Discussion

Greediness. QBFSYNT is greedy in exploiting implementation freedom. When synthesizing a circuit for one control signal c_j, QBFSYNT ensures that *some* solution for the remaining control signals still exists. However, the algorithm does not specifically attempt to retain implementation freedom for the remaining control signals. This can have the effect that the signals synthesized early have a small implementation, which is found after only a few refinements. Yet, for the signals synthesized later, the implementation freedom may already be “exhausted” and large implementations may be produced after many refinements. Consequently, the performance may also strongly depend on the order in which control signals are processed. This is similar to the standard COFSYNT procedure, but different from the QBF certification approach from Section 3.2.1, which computes circuits for all control signals simultaneously.

Independence of symbolic representation. In contrast to COFSYNT and the QBF certification approach, the QBF-based learning approach is rather independent of the symbolic strategy representation and the applied reasoning engine. Only the concrete counterexamples computed by Line 6 may differ, and our experience in trying to develop heuristics for computing good counterexamples indicates that one counterexample is usually just as good as any other. Consequently, the number of iterations and the resulting circuit will be similar, independent of whether the strategy formula is encoded efficiently or not. When implemented using BDDs, the variable ordering has little impact on these metrics too.

Circuit depth. Another advantage of the QBF-based CNF learning algorithm presented in this section is that the produced circuits have a low depth. This can be an important property because the circuit depth determines the maximum clock frequency with which the circuit can be operated. The formulas F_j defining the control signals c_j are computed in CNF. When these formulas are transformed into circuits in the straightforward way, this yields circuits with a depth of at most 3: every signal $\overline{x}, \overline{i}$ needs to pass at most one inverter, one OR-gate and one AND-gate. Depending on the gates available in the standard cell library, it may not be feasible to realize the circuit in this straightforward way. However, our experiments [82] with a simplistic standard cell library suggest that the circuit depth is usually much lower than when using the standard COFSYNT procedure with BDDs.

Algorithm 3.9 INTERPOLSYNT [124]: Synthesizing circuits from strategies using interpolation.

```
 1: procedure INTERPOLSYNT($S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$)
 2:     $\overline{c}_a := \overline{c}, \quad \overline{c}_b := \emptyset$
 3:     for all $j$ from $|\overline{c}|$ to $1$ do
 4:         $\overline{c}_a := \overline{c}_a \setminus \{c_j\}$
 5:         $M_1(\overline{x}, \overline{i}, \overline{c}_a) := \big(\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}')\big) \wedge \big(\neg\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}')\big)$
 6:         $M_0(\overline{x}, \overline{i}, \overline{c}_a) := \big(\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}')\big) \wedge \big(\neg\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}')\big)$
 7:         $F_j(\overline{x}, \overline{i}, \overline{c}_a) :=$ INTERPOL$\big(M_1(\overline{x}, \overline{i}, \overline{c}_a), M_0(\overline{x}, \overline{i}, \overline{c}_a)\big)$
 8:         DUMPCIRCUIT$\big(c_j, F_j(\overline{x}, \overline{i}, \overline{c}_a)\big)$
 9:         $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') := S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(c_j \leftrightarrow F_j(\overline{x}, \overline{i}, \overline{c}_a)\big)$
10:         $\overline{c}_b := \overline{c}_b \cup \{c_j\}$
11:     end for
12: end procedure
```

3.2.3 Interpolation

Jiang et al. [124] present an interpolation-based approach to synthesize circuits from strategies. Similar to the cofactor-based approach presented in Algorithm 2.2 and the QBF-based learning approach from Algorithm 3.7, it computes circuits for one control signal $c_j \in \overline{c}$ after the other. However, in contrast to these previous algorithms, the interpolation-based approach avoids quantifier alternations by temporarily considering other control signals for which no circuits have been computed yet as if they were inputs.

We will define the approach by Jiang et al. [124] as an algorithm for our setting in Section 3.2.3.1. After that, we will present optimizations and an efficient realization for safety specifications. In Section 3.2.4, we will furthermore combine the approach with query learning.

3.2.3.1 Basic Algorithm

The procedure INTERPOLSYNT in Algorithm 3.9 illustrates the approach by Jiang et al. [124] in our setting. As before, the input is a strategy formula $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$. The procedure does not return any result but directly dumps the produced circuits defining $\overline{c}$.

Variable dependencies. Similar to QBFSYNT in Algorithm 3.7, the variables $\overline{c} = (c_1, \dots, c_n)$ are split into three groups $\overline{c}_a, c_j, \overline{c}_b$. Here, c_j is the variable for which a circuit is computed in the current iteration. The algorithm starts with the last control signal c_n and proceeds with decreasing indices.[5] Line 10 makes sure that the variable vector $\overline{c}_b$ contains all control variables for which a circuit has been computed in some previous iteration. Finally, $\overline{c}_a$ contains all control variables for which a circuit needs to be computed in one of the following iterations. The variables in $\overline{c}_a$ are treated as if they were inputs. That is, the circuit defining c_j may not only reference variables from $\overline{x}$ and $\overline{i}$, but also all c_k with $k < j$ for which no circuit has been computed yet. This is illustrated in Figure 3.12: c_n can also take all signals $c_1, \dots, c_{n-1}$ as input, the circuit for c_{n-1} can also take $c_1, \dots, c_{n-2}$ as input, etc. Finally, c_1 cannot depend on any other variables of $\overline{c}$. This ensures that there are no circular dependencies. Furthermore, when the circuits for all $c_j \in \overline{c}$ are built together, the signals $\overline{c}$ effectively depend on $\overline{x}$ and $\overline{i}$ only.

Definition of M_1 and M_0. Let $\overline{d} = \overline{x} \cup \overline{i} \cup \overline{c}_a$ be the vector of all variables on which the current control signal c_j may depend. Line 5 of INTERPOLSYNT computes $M_1(\overline{d})$, which characterizes the set of all $\overline{d}$-assignments for which c_j must be true. This is done as follows. The subformula $C_1(\overline{d}) = \exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}')$ characterizes the set of all $\overline{d}$-assignments for which $c_j = \mathsf{true}$ is allowed by S. This is essentially the positive cofactor of S regarding c_j, but the variables $\overline{c}_b, \overline{x}'$ are also quantified existentially, which means that their concrete value is irrelevant as long as some value exists. Similarly, the subformula $C_0(\overline{d}) = \exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}')$ characterizes the set of all $\overline{d}$-assignments for

[5] The order is actually irrelevant, but fixing some order simplifies the discussion.

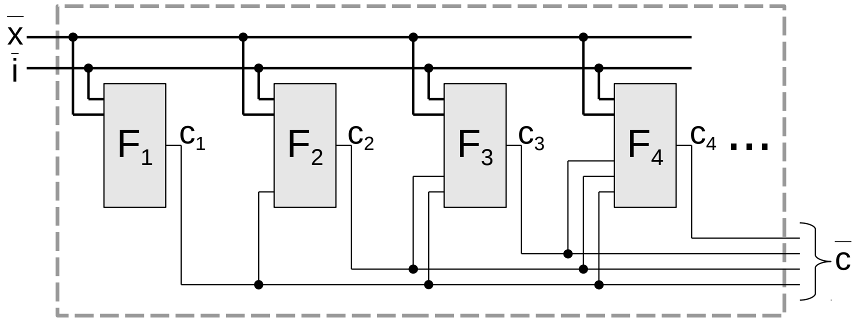


Figure 3.12: Variable dependencies in interpolation-based circuit synthesis. The first control signal c_1 can only depend on $\overline{x}$ and $\overline{i}$. The next signal c_2 can also depend on c_1. The signal c_3 can also depend on c_1 and c_2, and so on. Yet, if all circuits F_j are combined, the control signals $\overline{c}$ will effectively depend on $\overline{x}$ and $\overline{i}$ only.

which $c_j = \mathsf{false}$ is allowed by S. Hence, M_1 represents the set of all $\overline{d}$-assignments for which true is allowed, but false is not allowed. Analogously, Line 6 computes the formula M_0, which characterizes the $\overline{d}$-assignments for which c_j must be false. In principle, M_1 and M_0 can easily be transformed into a propositional CNF formula by renaming or expanding the existentially quantified variables. An efficient solution to do so will be presented in Section 3.2.3.3, but for now we focus on understandability rather than efficiency.

Differences to QBFSYNT. Note that the procedure QBFSYNT from Algorithm 3.7 computes M_1 and M_0 differently in two respects. First, $M_1(\overline{x}, \overline{i})$ and $M_0(\overline{x}, \overline{i})$ do not contain $\overline{c}_a$ as free variables in QBFSYNT. Second, $M_1(\overline{x}, \overline{i})$ is computed as $\neg C_0(\overline{x}, \overline{i})$ in QBFSYNT instead of $C_1(\overline{d}) \wedge \neg C_0(\overline{d})$ (and similar for $M_0(\overline{x}, \overline{i})$). The additional conjunction with $C_1(\overline{d})$ in INTERPOLSYNT is necessary for the following reason. We have that $\neg C_0(\overline{x}, \overline{i}) \rightarrow C_1(\overline{x}, \overline{i})$ and $\neg C_1(\overline{x}, \overline{i}) \rightarrow C_0(\overline{x}, \overline{i})$ in QBFSYNT because $\forall \overline{x}, \overline{i} : \exists \overline{c}, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}, \overline{x}')$ is guaranteed by the strategy. In other words, for every $(\overline{x}, \overline{i})$-assignment, any control signal c_j can either be true or false (or both). Hence, the additional conjunct $C_1(\overline{x}, \overline{i})$ would be of no use in $M_1(\overline{x}, \overline{i}) = \neg C_0(\overline{x}, \overline{i})$ as defined by QBFSYNT, because it is implied anyway. Yet, $\neg C_0(\overline{d}) \rightarrow C_1(\overline{d})$ and $\neg C_1(\overline{d}) \rightarrow C_0(\overline{d})$ do *not* hold in INTERPOLSYNT: there may be some $\overline{d}$-assignment for which neither $c_j = \mathsf{true}$ nor $c_j = \mathsf{false}$ is allowed by the strategy. The reason is that we also consider the signals $\overline{c}_a$ as if they were inputs, but $\forall \overline{x}, \overline{i}, \overline{c}_a : \exists c_j, \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, c_j, \overline{c}_b, \overline{x}')$ does not hold in general. For $\overline{d}$-assignments for which neither $c_j = \mathsf{true}$ nor $c_j = \mathsf{false}$ is allowed, the definition of $M_1 = C_1(\overline{d}) \wedge \neg C_0(\overline{d})$ and $M_0 = C_0(\overline{d}) \wedge \neg C_1(\overline{d})$ allows both values for c_j. This is justified by the fact that the circuits synthesized for $\overline{c}_a$ in subsequent iterations will make sure that such $\overline{d}$-assignments will never occur as input of the circuit defining c_j. We refer to Jiang et al. [124] for details on this technical subtlety.

Interpolation. The conjunction $M_1(\overline{d}) \wedge M_0(\overline{d}) = C_1(\overline{d}) \wedge \neg C_0(\overline{d}) \wedge C_0(\overline{d}) \wedge \neg C_1(\overline{d})$ is trivially unsatisfiable, so an interpolant $F_j(\overline{d})$ can be computed in Line 7. The properties of an interpolant (see Section 2.2.1) ensure that $M_1 \rightarrow F_j \rightarrow \neg M_0$. The first implication means that F_j is true whenever c_j must be true. The second implication means that if F_j is true, then c_j does not have to false. This means that $F_j(\overline{d})$ is a proper implementation for c_j.

Circuit construction and resubstitution. The remaining steps of the INTERPOLSYNT procedure are the same as for CofSynt (Algorithm 2.2) and QbfSynt (Algorithm 3.7). Line 8 constructs a circuit which sets $c_j = \mathsf{true}$ if an only if $F_j(\overline{x}, \overline{i}, \overline{c}_a)$ evaluates to true. Finally, Line 9 refines the strategy formula S with the concrete implementation for c_j.

Auxiliary variables. If the strategy formula S is defined using auxiliary variables, these can all be put into $\overline{c}_b$. This also applies to auxiliary variables that may be introduced in the resubstitution in Line 9.

Variations. Jiang et al. [124] propose to perform a second pass over all control signals, where the circuits for all c_j are recomputed using interpolation, while fixing the implementation for the other control signals. This has the potential for producing smaller circuits because the recomputed interpolants can now rely on some concrete realization for the other control signals. However, in preliminary experiments for our setting, this second pass did not result in considerable circuit size improvements (but rather increased the circuit size for many cases). Since such a second pass also increases the computation time, we do not perform it. Jiang et al. [124] also propose a second interpolation-based approach which does not treat other control signals as if they were inputs but rather quantifies them universally and applies universal expansion to eliminate the quantifiers. However, this can blow up the formula size significantly. Preliminary experiments with this second approach were not promising in our setting either.

3.2.3.2 Dependency Optimization

For some specifications, the performance of INTERPOLSYNT strongly depends on the order in which the control signals $\overline{c} = (c_1, \ldots, c_n)$ are processed. One reason is that this order defines which signal c_j may depend on which other signals c_k. The aim of the optimization presented in this section is to increase the set of variables on which a certain signal c_j can depend. This increases the freedom for the interpolation procedure (the interpolant F_j may still choose the ignore the additional signals) and can lead to smaller interpolants and shorter execution times.

Basic idea. The basic idea is as follows. As illustrated in Figure 3.12, the interpolant F_n computed first can reference all other control signals $c_1, \ldots, c_{n-1}$. The interpolant F_{n-1} computed in the second iteration cannot depend on c_n, though. The reason is that F_n, which defines c_n, could in turn reference c_{n-1}, which would result in a circular dependency. Yet, the concrete interpolant F_n may choose to ignore c_{n-1} completely. In this case, F_{n-1} *can* in fact be allowed to reference c_n. The reason is that there is no danger to introduce a circular dependency — the result would be the same as if c_n and c_{n-1} would have been processed by INTERPOLSYNT in reverse order.

Realization. In the iteration synthesizing a solution for c_j, we analyze which other signals c_k with $k > j$ do not transitively depend on c_j. This is done on a syntactic level by checking if c_j occurs in the fan-in cone of c_k when the circuits for $F_{j+1}, \ldots, F_n$ are combined. If c_j does not appear in the fan-in cone of c_k, then c_k is moved (temporarily) from $\overline{c}_b$ to $\overline{c}_a$. Thus, $F_j(\overline{x}, \overline{i}, \overline{c}_a)$ can reference c_k.

Dependencies on auxiliary variables. Depending on the realization of INTERPOL, the computed interpolants $F_j(\overline{x}, \overline{i}, \overline{c}_a)$ may be represented using auxiliary variables (e.g., introduced by a Tseitin-transformation [208]) that act as abbreviation for some subformulas over $\overline{x}$, $\overline{i}$ and $\overline{c}_a$. As mentioned in the previous subsection, all auxiliary variables are put into $\overline{c}_b$, so they cannot be referenced by the computed interpolants by default. However, the dependency analysis cannot only be performed for the final output of each F_k with $k > j$, but also on their auxiliary variables: if the current c_j does not appear in the fan-in cone of some auxiliary variable t, then t can be moved from $\overline{c}_b$ to $\overline{c}_a$.

3.2.3.3 Efficient Implementation for Safety Synthesis Problems

The procedure SAFEINTERPOLSYNT in Algorithm 3.10 shows an efficient implementation of INTERPOLSYNT if the winning strategy is defined via a winning region (or winning area) $W(\overline{x})$ of a safety specification. The dependency optimization is not included for the sake of readability.

Computation of M_1 and M_0. With $S(\overline{x}, \overline{i}, \overline{c}, \overline{x}') = T(\overline{x}, \overline{i}, \overline{c}, \overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)$, we can apply the following transformations to compute a more compact CNF for M_1.

$$\begin{aligned}
M_1(\overline{x}, \overline{i}, \overline{c}_a) =& \big(\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}')\big) \wedge \big(\neg\exists \overline{c}_b, \overline{x}' : S(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}')\big)\\
=& \Big(\exists \overline{c}_b, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)\Big) \wedge\\
& \Big(\neg\exists \overline{c}_b, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)\Big)\\
=& \Big(\exists \overline{c}_b, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{true}, \overline{c}_b, \overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)\Big) \wedge\\
& \Big(\forall \overline{c}_b, \overline{x}' : T(\overline{x}, \overline{i}, \overline{c}_a, \mathsf{false}, \overline{c}_b, \overline{x}') \rightarrow \big(W(\overline{x}) \wedge \neg W(\overline{x}')\big)\Big)
\end{aligned}$$

Algorithm 3.10 SafeInterpolSynt: Synthesizing circuits from winning areas using interpolation.

```
 1: procedure SafeInterpolSynt($T(\overline{x},\overline{i},\overline{c},\overline{x}')$, $W(\overline{x})$)
 2:     $T'(\overline{x},\overline{i},\overline{c},\overline{x}') := T(\overline{x},\overline{i},\overline{c},\overline{x}')$,    $\overline{c}_a := \overline{c}$,    $\overline{c}_b := \emptyset$
 3:     for all $j$ from $|\overline{c}|$ to 1 do
 4:         $\overline{c}_a := \overline{c}_a \setminus \{c_j\}$
 5:         $\overline{c}_{b1}, \overline{c}_{b2}, \overline{c}_{b3}, \overline{c}_{b4} :=$ create4FreshCopies($\overline{c}_b$)
 6:         $\overline{x}'_1, \overline{x}'_2, \overline{x}'_3, \overline{x}'_4 :=$ create4FreshCopies($\overline{x}'$)
 7:         $M'_1(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b1},\overline{c}_{b2},\overline{x}'_1,\overline{x}'_2) := T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b1},\overline{x}'_1) \wedge W(\overline{x}'_1) \wedge$
                $T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b2},\overline{x}'_2) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_2)$
 8:         $M'_0(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b3},\overline{c}_{b4},\overline{x}'_3,\overline{x}'_4) := T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b3},\overline{x}'_3) \wedge W(\overline{x}'_3) \wedge$
                $T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b4},\overline{x}'_4) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_4)$
 9:         $F_j(\overline{x},\overline{i},\overline{c}_a) := \textsc{Interpol}\big(M'_1(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b1},\overline{c}_{b2},\overline{x}'_1,\overline{x}'_2), M'_0(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b3},\overline{c}_{b4},\overline{x}'_3,\overline{x}'_4)\big)$
10:         dumpCircuit$\big(c_j, F_j(\overline{x},\overline{i},\overline{c}_a)\big)$
11:         $T'(\overline{x},\overline{i},\overline{c},\overline{x}') := T'(\overline{x},\overline{i},\overline{c},\overline{x}') \wedge \big(c_j \leftrightarrow F_j(\overline{x},\overline{i},\overline{c}_a)\big)$
12:         $\overline{c}_b := \overline{c}_b \cup \{c_j\}$
13:     end for
14: end procedure
```

That is, the negation turns the existential quantification over $\overline{c}_b, \overline{x}'$ into a universal one. Yet, just like SafeQbfSynt (Algorithm 3.8), SafeInterpolSynt also keeps a copy T' of the transition relation T that defines all variables in $\overline{c}_b$ and $\overline{x}'$ uniquely based on the other variables. For the variables $\overline{x}'$, this holds initially. For $\overline{c}_b$, this is ensured by Line 11. Thus, by using T' instead of T and by applying the one-point rule (Equation 2.2), the universal quantification can be turned into an existential one:

$$\begin{aligned} M_1(\overline{x},\overline{i},\overline{c}_a) = &\Big(\exists \overline{c}_b, \overline{x}' : T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_b,\overline{x}') \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}')\big)\Big) \wedge \\ &\Big(\exists \overline{c}_b, \overline{x}' : T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_b,\overline{x}') \wedge W(\overline{x}) \wedge \neg W(\overline{x}')\Big) \end{aligned}$$

By renaming the variables $\overline{c}_b$ and $\overline{x}'$, the two subformulas can be merged into one block of quantifiers:

$$\begin{aligned} M_1(\overline{x},\overline{i},\overline{c}_a) = \exists \overline{c}_{b1}, \overline{c}_{b2}, \overline{x}'_1, \overline{x}'_2 :\ & T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b1},\overline{x}'_1) \wedge \big(\neg W(\overline{x}) \vee W(\overline{x}'_1)\big) \wedge \\ & T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b2},\overline{x}'_2) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_2) \end{aligned}$$

Finally, $\big(\neg W(\overline{x}) \vee W(\overline{x}'_1)\big) \wedge W(\overline{x})$ can be simplified to $W(\overline{x}) \wedge W(\overline{x}'_1)$, which is fortunate because negations and disjunctions are expensive to perform in CNF. This gives $M_1(\overline{x},\overline{i},\overline{c}_a) =$

$$\exists \overline{c}_{b1}, \overline{c}_{b2}, \overline{x}'_1, \overline{x}'_2 : T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b1},\overline{x}'_1) \wedge W(\overline{x}'_1) \wedge T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b2},\overline{x}'_2) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_2).$$

In SafeInterpolSynt, the existential quantification is not applied. Instead, the variables $\overline{c}_{b1}$, $\overline{c}_{b2}$, $\overline{x}'_1$, $\overline{x}'_2$ occur freely in M'_1. Similarly, other fresh copies $\overline{c}_{b3}, \overline{c}_{b4}, \overline{x}'_3, \overline{x}'_4$ of the same variables occur freely in M'_0. The properties of an interpolant (see Section 2.2.1) ensure that F_j, computed in Line 9, can only reference the variables $\overline{x}, \overline{i}, \overline{c}_a$ occurring both in M'_1 and in M'_0. Hence, these free variables cannot be referenced in the resulting circuit.

CNF conversion. The formulas in Line 7 and 8 contain only conjunctions. Most of our methods to compute a winning region or a winning area produce $W(\overline{x})$ in CNF. Hence, just as for QBF certification and QBF-based CNF learning, we only need to compute a CNF representation of T' and $\neg W(\overline{x}')$.

Simplification of interpolants. The computed interpolants F_j refine T' in Line 11. Hence, complicated representations of F_j result in more complicated formulas for T', which can increase the time needed for interpolation quite significantly (and may result in even more complicated formulas for the subsequent interpolants). Besides optimizing the final circuit regarding size, we therefore also optimize every single interpolant using the tool ABC [42] after it has been computed.

3.2.3.4 Discussion

Exploiting implementation freedom. INTERPOLSYNT is rather conservative in exploiting implementation freedom when computing a circuit for some control signal c_j: to the extend where this is feasible, the circuit F_j defining c_j will work for *any* realization of the control signals $\overline{c}_a$ that have not been synthesized yet. The reason is that the variables of $\overline{c}_a$ are handled as if they were inputs. This stands in contrast to QBFSYNT, which is more greedy by exploiting implementation freedom as long as *some* solution for the other signals still exists. Both strategies have their advantages. Preserving implementation freedom can result in smaller circuits for control signals that are synthesized later. The greedy strategy can be better in preventing that implementation freedom is left unexploited.

Dependencies between control signals. In contrast to COFSYNT and QBFSYNT, INTERPOLSYNT constructs a circuit in such a way that the implementation for one control signal can be reused in the implementation of others (see Figure 3.12). This can result in a smaller total circuit size. As an extreme example, one control signal c_j could be required to be an exact copy of some other control signal c_k. While INTERPOLSYNT may find the implementation $c_j = c_k$ quickly, both COFSYNT and QBFSYNT would have to construct the same (potentially complicated) circuit based on the variables $\overline{x}$ and $\overline{i}$ twice. The circuit optimization techniques we apply as a postprocessing step may optimize one copy away, so the final circuit may actually be the same. Nevertheless, computing the same circuit twice is at least a waste of computing resources.

Dependence on the interpolation procedure. With INTERPOLSYNT, the size of the resulting circuits strongly depends on the ability of the interpolation procedure INTERPOL to exploit the freedom between M_1 and $\neg M_0$. When the interpolant is computed from an unsatisfiability proof returned by a SAT solver, we must rely on the heuristics in the solver to yield a compact proof that can be used to derive a simple interpolant, which can then be implemented in a small circuit. In contrast, QBFSYNT is more independent of the underlying reasoning engine. The next section will present an approach to reduce this dependency of INTERPOLSYNT on the underlying solver.

3.2.4 Query Learning Based on SAT Solving

In this section, we combine query learning with the idea by Jiang et al. [124] to temporarily treat control signals as if they were inputs. This eliminates the need for universal quantification and allows us to implement the query learning approach from Section 3.2.2 with a SAT solver instead of a QBF solver.

In the following subsection, we will again present a solution based on CNF learning. Applying other learning algorithms from our FMCAD'12 [82] publication is possible, but imposes more overhead for encoding formula parts into CNF. After introducing the basic algorithm, we will again present an efficient realization for safety synthesis problems and discuss the differences to the other algorithms.

3.2.4.1 CNF Learning Based on SAT Solving

In Section 3.2.2, we have already discussed that query learning can be used as a special interpolation procedure if different formulas are used for counterexample computation and generalization. While Section 3.2.2 used this idea to compute interpolants between quantified formulas using a QBF solver, we use it here to compute interpolants for propositional formulas using CNF learning.

Algorithm. We keep the basic structure of the INTERPOLSYNT procedure from Algorithm 3.9, but replace the call to INTERPOL in Line 7 by a call to CNFINTERPOL, which is defined in Algorithm 3.11. The interface of CNFINTERPOL is the same as that of any interpolation procedure: given two formulas $M_1(\overline{d}, \overline{t}_1)$ and $M_0(\overline{d}, \overline{t}_0)$ such that $M_1 \wedge M_0$ is unsatisfiable, it returns a formula $F(\overline{d})$ over the shared variables $\overline{d}$ such that $M_1 \rightarrow F \rightarrow \neg M_0$. The implementation of CNFINTERPOL is simple. It starts with the initial approximation $F = \mathsf{true}$ and enforces the invariant $M_1 \rightarrow F$. Line 3 checks if $F \rightarrow \neg M_0$, which is the case if and only if $F \wedge M_0$ is unsatisfiable. If so, then $M_1 \rightarrow F \rightarrow \neg M_0$ holds, so the loop terminates and F is returned as result. Otherwise a counterexample $\mathbf{d} \models F \wedge M_0$ is extracted for which

Algorithm 3.11 CNFINTERPOL: Computing an interpolant using CNF learning with a SAT solver.

```
1: procedure CNFINTERPOL$\big(M_1(\overline{d},\overline{t}_1), M_0(\overline{d},\overline{t}_0)\big)$, returns: A CNF $F(\overline{d})$ with $M_1 \rightarrow F \rightarrow \neg M_0$
2:     $F(\overline{d}) := \mathsf{true}$
3:     while sat in $(\mathsf{sat}, \mathbf{d}) :=$ PROPSATMODEL$\big(M_0(\overline{d},\overline{t}_0) \wedge F(\overline{d})\big)$ do
4:         $F(\overline{d}) := F(\overline{d}) \wedge \neg$PROPMINUNSATCORE$\big(\mathbf{d}, M_1(\overline{d},\overline{t}_1)\big)$
5:     end while
6:     return $F(\overline{d})$
7: end procedure
```

F is true but must be false. The computation of the unsatisfiable core in Line 4 generalizes the cube $\mathbf{d}$ by dropping literals as long as $\mathbf{d}$ does not intersect with M_1. Consequently, the update of F in Line 4 preserves the invariant $M_1 \rightarrow F$ and resolves the counterexample.

Exploiting freedom. As in QBFSYNT (Algorithm 3.7) using M_0 in counterexample computation makes sure that refinements of F are only triggered if some $\overline{d}$-assignment $\mathbf{d}$ *must* be mapped to false. Using M_1 in counterexample generalization entails that other $\overline{d}$-assignments are also mapped to false as long as they *can* be mapped to false. Using "can" instead of "must" during generalization potentially eliminates more counterexamples before they are actually encountered by Line 3.

3.2.4.2 Efficient Implementation for Safety Synthesis Problems

Following the transformations presented for SAFEINTERPOLSYNT in Section 3.2.3.3, we have that CNFINTERPOL is called with

$$M_1(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b1},\overline{c}_{b2},\overline{x}'_1,\overline{x}'_2) = \\ T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b1},\overline{x}'_1) \wedge W(\overline{x}'_1) \wedge T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b2},\overline{x}'_2) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_2) \text{ and}$$

$$M_0(\overline{x},\overline{i},\overline{c}_a,\overline{c}_{b3},\overline{c}_{b4},\overline{x}'_3,\overline{x}'_4) = \\ T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{false},\overline{c}_{b3},\overline{x}'_3) \wedge W(\overline{x}'_3) \wedge T'(\overline{x},\overline{i},\overline{c}_a,\mathsf{true},\overline{c}_{b4},\overline{x}'_4) \wedge W(\overline{x}) \wedge \neg W(\overline{x}'_4).$$

Since CNFINTERPOL itself does not contain any negations nor disjunctions, only $\neg W(\overline{x}')$ needs to be transformed into CNF.

Dependency optimization. We can apply the dependency optimization presented in Section 3.2.3.2. However, on top of allowing dependencies on other control signals, we also allow dependencies on auxiliary variables that are used for defining the transition relation T' as long as this does not result in circular dependencies.

Incremental solving. CNFINTERPOL is well suited for incremental SAT solving. A simple solution uses two solver instances, which are initialized whenever CNFINTERPOL is called. The first solver instance stores $M_0 \wedge F$ and is used for Line 3. The second one stores M_1 and is used for Line 4. A more radical solution uses only one solver instance throughout *all* calls to CNFINTERPOL. Note that M_1 differs from M_0 only by having c_j (in two copies) set to different truth constants. Hence, switching between M_1 and M_0 can be achieved by setting (the two copies of) c_j differently with assumption literals. Furthermore, the clauses of some F_j are all disjoined with some fresh activation variable a_j before they are asserted in the solver. This way, F_j can be enabled or disabled by setting the assumption literal $\neg a_j$ or a_j, respectively. Finally, T' changes between major iterations of SAFEINTERPOLSYNT (see Line 11). However, additional constraints are only added in this update, so this does not pose any challenge for incremental solving.

Minimizing the final solution. Recall from the interpolation-based method from Section 3.2.3 that a second pass over all control signals can be performed, in which the circuits for all c_j are recomputed while the implementation for the other control signals is fixed. In principle, this has the potential for reducing the circuit size because the recomputed circuits can now rely on some concrete realization for the other control signals. The same idea can also be applied in our SAT solver based CNF learning approach. However, similar to interpolation, recomputing individual circuits by learning them from scratch did not result in circuit size reductions, but more often in circuit size increases in our experiments. Yet, instead of recomputing a circuit from scratch, we can also start with the existing solution F_j, which is given as a CNF formula, and simplify it by dropping literals and clauses as long as correctness is still preserved. The idea is similar to COMPRESSCNF (Algorithm 3.2), but the simplification is not equivalence preserving but only correctness preserving. We propose to postprocess all F_j in the order of decreasing j. The reason is that F_n was computed first, without any knowledge about the implementation of the other F_j. Hence, intuitively, F_n has the greatest potential for simplifications relying on the concrete realization of all other F_j. Each F_j satisfies $M_1 \rightarrow F_j \rightarrow \neg M_0$ initially, where M_1 and M_0 are now defined using the concrete implementation for the other F_k. We propose to simplify each F_j in two phases. The first phase drops literals from clauses of F_j as long as $M_1 \rightarrow F_j$ is preserved (because dropping literals can make F_j only stronger). Similar to COMPRESSCNF, this can be realized by computing unsatisfiable cores, utilizing incremental SAT solving. The second phase drops clauses from F_j, starting with the longest ones, as long as $F_j \rightarrow \neg M_0$ is still preserved (because dropping clauses can make F_j only weaker). Since we only drop literals and clauses from the existing implementations, this postprocessing can only make the resulting circuits smaller but never larger.

3.2.4.3 Discussion

The SAT solver based CNF learning approach is very similar to the interpolation-based method from the previous section, and thus inherits most of its strength and weaknesses. However, using the learning algorithm instead of interpolation makes the approach less dependent on the underlying solver. This is similar to QBFSYNT. Also similar to QBFSYNT is the fact that individual circuits are computed as formulas in CNF. However, because the individual circuits are cascaded as illustrated in Figure 3.12, the final circuit depth will in general be higher than that of circuits produced by QBFSYNT. Still, the circuit depths can be expected to be lower compared to INTERPOLSYNT in most cases. The reason is that interpolants derived from an unsatisfiability proof can have a depth that is much higher than 3, and the procedure for building the individual circuits together is the same.

3.2.5 Parallelization

We have already discussed that different methods for circuit synthesis have different characteristics. The experimental results in Section 3.3 will indicate that this results in different methods and optimizations performing well on different classes of benchmarks. Similar to strategy computation (see Section 3.1.7) we thus propose a parallelization that executes different methods and optimizations in different threads. The aim is to combine the strengths and compensate the weaknesses of the individual methods.

Realization. In contrast to Section 3.1.7, our parallelization for synthesizing circuits from strategies follows a rather simple portfolio approach, where each thread solves the circuit synthesis problem without any information from other threads. The first thread implements the SAT solver based learning algorithm from Section 3.2.4 with the dependency optimization from Section 3.2.3.2. If our parallelization is executed with two threads, the second thread performs QBF-based CNF learning (Section 3.2.2) with incremental QBF solving. If executed with three threads, the third thread again performs learning using a SAT, but without the dependency optimization.

Heuristics. In order to achieve a good balance between low execution time and small circuits, the user can inform our parallelization about a timeout. A heuristic then uses this information to decide whether to perform a minimization of the final solution, as explained in Section 3.2.4.2, or not.[6] Furthermore, if one thread finishes, it does not stop the other threads immediately but only if the user-defined

[6] For the experiments, we used a very conservative heuristic: if the remaining time available is more than 10 times the time used so far for computing a circuit from the strategy, then the minimization of the final solution will be performed.

timeout is approaching or the ratio between waiting time and working time exceeds a certain threshold (0.25 in our experiments). The reason is that, from all threads that terminated, we finally select the circuit with the lowest number of gates. Hence, even if one thread has already found a solution, waiting for other threads to finish their computation can be beneficial for the final circuit size.

Alternatives. As for strategy computation, there is a plethora of possibilities to combine different methods while sharing information in a more fine-grained way. Since most of the methods compute circuits for one control signal after the other, the final solutions for each control signal can be exchanged. Each thread can then continue with the smallest solution that has been found for the respective signal. Since several methods are based on counterexample-guided refinements of solution candidates, the respective threads can also directly exchange counterexamples and the corresponding blocking clauses. Furthermore, it can be beneficial to have different threads synthesizing circuits for control signals in different order. We leave an exploration of such fine-grained parallelization approaches for future work.

3.3 Experimental Results

In this section, we will first sketch our implementation of the SAT-based synthesis algorithms introduced so far. After that, we will describe benchmarks that will be used in our experimental evaluation (Section 3.3.2). The core of this section is formed by our performance evaluation for computing strategies (Section 3.3.3) and for constructing circuits from strategies (Section 3.3.4). The section concludes with a discussion of the central results (Section 3.3.5).

3.3.1 Implementation

We have implemented the synthesis methods presented in Section 3.1 and Section 3.2 in a synthesis tool called Demiurge. It is written in C++ and compatible with the rules for the SyntComp [117] synthesis competition. Demiurge has won two gold medals in this synthesis competition: one in 2014 and one in 2015, both in the parallel synthesis track. The input of Demiurge is a safety specification in AIGER[7] format. The synthesis result is a circuit in AIGER format as well. Since the synthesis process does not involve any interaction with the user except for setting parameters, Demiurge does not come with a GUI, but is started from the command-line. So far, our synthesis tool has only been tested on Linux operating systems. Demiurge is freely available under the GNU Lesser General Public License[8] version 3, and can be downloaded from

`https://www.iaik.tugraz.at/content/research/opensource/demiurge/`.

All experiments presented in this thesis have been performed using version `1.2.0`. The downloadable archive contains all scripts to reproduce the experiments, as well as spreadsheets with more detailed data (such as execution times for individual steps of the algorithms, numbers of iterations, etc.).

Architecture. The architecture of Demiurge is outlined in Figure 3.13. The AIG2CNF module parses the specification into CNF formulas representing the transition relation T and the set of safe states P. Only one initial state is allowed in the input format, so the initial states I in our definition of a safety specification are represented as a minterm. Next, the back end selected by the user via command-line options is executed. The back ends mostly differ in their method for computing the winning region (or a winning area), and can be parameterized with a method for computing the circuit from the induced winning strategy. Furthermore, the back ends can be configured with options to enable or disable optimizations or optional steps. The back ends can access a number of different solvers via uniform interfaces. That is, multiple SAT solvers can be accessed via the same abstract interface, which hides the concrete solver from the application. Various QBF solvers are accessible via a second interface

[7] `http://fmv.jku.at/aiger/` (last visit on 2015-08-01).

[8] Contact the authors if you want to obtain a copy of the tool under a different license.

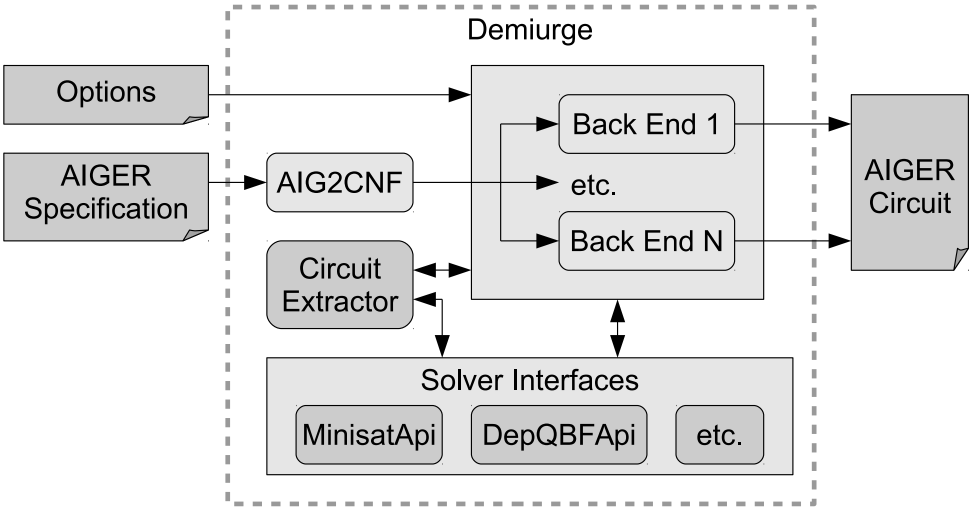


Figure 3.13: Architecture of the SAT-based synthesis tool Demiurge. The input is a safety specification in AIGER format. The output is an AIGER circuit. Different back ends implement different methods to compute a winning region. They are parameterized with a method for computing a circuit from the induced winning strategy. Back ends can access third-party solvers via uniform interfaces.

(which is similar to the interface for SAT solvers). The concrete solvers that shall be used are again configured via command-line options. Due to this extensible architecture, Demiurge can also be seen as a framework for implementing new synthesis algorithms or optimizations with low effort: A lot of infrastructure such as the parser, interfaces to solvers and entire synthesis steps (like computing a circuit from a strategy) can be reused.

External tools. In version `1.2.0`, Demiurge has interfaces to

- the SAT solver MiniSat [80] in version `2.2.0` via its API,
- the SAT solver PicoSAT [22] in version `960` via its API,
- the SAT solver Lingeling [23] in version `ayv` via its API,
- the QBF solver DepQBF [155, 157] in version `3.04` via its API, both with and without preprocessing by Bloqqer [26, 189] version `34`,
- the QBF solver RAReQS [120] in version `1.1` via a self-made API,
- the QBF solver QuBE [98] in version `7.2` with communication via files,
- the tool ABC [42] (commit `d3db71b`) for optimizing AIGER circuits with communication via files, and
- the first-order theorem prover iProver [146] in version `1.0` with communication via files.

3.3.2 Benchmarks

We used benchmarks from the SyntComp 2014 [117] benchmark set[9] to evaluate the performance of our different methods to compute strategies as well as circuits implementing these strategies. Most of the benchmarks are parameterized. In the following, we briefly summarize their main characteristics as far as this is helpful for interpreting the performance results. The size of the benchmarks is summarized in Table 3.1. All in all, we included 350 benchmark instances, of which 40 instances are unrealizable.

The **add**ko benchmark specifies a combinational adder for two k-bit numbers. The parameter $o \in \{\mathtt{y}, \mathtt{n}\}$ indicates if the benchmark file has been optimized with ABC [42] for circuit size (value `y`) or not

[9] We used *all* benchmark instances from this set with two exceptions: From the **amba** and **genbuf** benchmarks, we did not select the unoptimized and the unrealizable instances to keep the number of instances manageable and balanced. Second, we also included a **driver** benchmark that is not contained in the SyntComp 2014 benchmark set.

Table 3.1: Summary of benchmark sizes. The suffix k means that the respective number is multiplied by 1000. The suffix M means a multiplication by one million. The last column lists the number of AND-inverter gates defining the transition relation T.

Name	parameter range	$\lvert\overline{x}\rvert$	$\lvert\overline{i}\rvert$	$\lvert\overline{c}\rvert$	Gates in T
addko	$k = 2$ to 20	2	$2 \cdot k$	k	17 to 365
mult$k$	$k = 2$ to 16	0	$2 \cdot k$	$2 \cdot k$	24 to 2450
cntko	$k = 2$ to 30	$k + 1$	1	1	11 to 450
mvko	$k = 2$ to 28	$k + 1$	$k - 1$	$k - 1$	10 to 469
bsko	$k = 8$ to 128	$k + 1$	$\mathrm{ld}(k)$	1	80 to 3202
stayko	$k = 2$ to 24	$k + 2$	k	$k + 1$	17 to 4104
ambakl	$k = 2$ to 10	28 to 76	$2 \cdot k + 3$	8 to 19	177 to 630
genbufkl	$k = 1$ to 16	21 to 73	$k + 4$	6 to 24	134 to 733
fact$mnkc$	special selection	20 to 54	10 to 40	8 to 12	122 to 594
movklm	$k = l = 8$ to 128	19 to 41	12 to 34	5	306 to 830
driverkl	$l = 5$ to 8	55 to 326	16 to 98	24 to 82	435 to 1942
demokl	$k = 1$ to 25	12 to 280	1 to 4	1 to 4	43 to 2055
gb$k$	$k = 1$ to 4	11k to 23k	4	4	867k to 1.7M
loadkl	$k = 2$ to 3	96 to 296	3 to 4	2 to 3	1092 to 3156
ltl2dbakl	$k = 1$ to 20	44 to 484	2 to 7	1	194 to 5482
ltl2dpa$k$	$k = 1$ to 18	44 to 340	1 to 3	2 to 4	191 to 3866

(value n). This benchmarks is realizable. Since it is mostly combinational, it challenges circuit synthesis more than strategy computation.

The **mult**k benchmark specifies a combinational multiplier for two k-bit numbers and, thus, has similar characteristics to **add**.

The **cnt**ko benchmark specifies a k-bit counter that must not reach its maximum value. At value $2^{k-1} - 1$, the counter can be reset if the only control signal is set to true. The parameter $o \in \{\mathtt{y}, \mathtt{n}\}$ again indicates if the benchmark was optimized. This benchmark is realizable and can be challenging for strategy computation because it may require many iterations to find the winning region. It is trivial for circuit synthesis, though, because hardwiring the only control signal to true suffices.

The **mv**ko benchmark also contains a k-bit counter that must not reach its maximum value. However, when the most significant counter bit is set, the counter can be reset if the XOR sum of all control signals is true. Hence, there exists an implementation that hardwires all control signals to constant values. Again, $o \in \{\mathtt{y}, \mathtt{n}\}$ indicates if the benchmark was optimized. The benchmark is realizable and can be challenging for circuit synthesis because it contains many interdependent control signals.

The realizable benchmark **bs**ko applies a barrel shifter to a k-bit register, which is initialized to some constant value and must never reach specific values. The amount of shifting is defined by uncontrollable inputs, but the shifting can be disabled with a control signal. Barrel shifters can be particularly challenging for BDDs.

The benchmark **stay**ko again contains a k-bit counter that must not reach its maximum value. Whether the counter is incremented or not depends on complicated logic, involving an arithmetic multiplication of the control signals with the uncontrollable inputs. Yet, when setting one specific control signal always to false, the specification is always satisfied. Hence, the crux with this benchmark is whether the algorithms can find and exploit this "backdoor".

The benchmark **amba**kl specifies an arbiter for ARM's AMBA AHB bus [33] with k bus masters. The parameter $l \in \{\mathtt{b}, \mathtt{c}, \mathtt{f}\}$ describes the method that has been used for transforming liveness properties in the original formulation of the benchmark [33] into safety properties. We refer to Jacobs et al. [117] for a description of these three transformations. All benchmark instances are available in an optimized and in an unoptimized form. Additionally, all benchmark instances are available in an unrealizable variant. However, since the performance difference between all these variants are rather small, we only ran our experiments with the realizable and optimized versions. This keeps the number of instances manageable.

The benchmark **genbuf**kl specifies a generalized buffer [33] connecting k senders to two receivers. The parameter $l \in \{\mathtt{b}, \mathtt{c}, \mathtt{f}\}$ is the same as for the **amba** benchmarks. Similar to **amba**, we only ran our experiments with the realizable and optimized versions in order to reduce the number of instances.

The **fact**$mnkc$ benchmark specifies a factory line with m tasks that need to be performed by two manipulation arms on a continuous stream of objects. The factory belt has n places and rotates every k cycles by one place, thereby delivering an object. The parameter c is a maximum number of errors in the setup of the processed objects that needs to be tolerated by the factory line. Some of the included benchmark instances are unrealizable.

The **mov**klm benchmark specifies a robot that has to move in a two-dimensional grid of $k \times l$ cells while avoiding collisions with a moving obstacle. By default, the obstacle can only move in every second step. However, at most m times, the obstacle can also move in consecutive time steps. For every grid size, our benchmark set contains an unrealizable and a realizable instance.

The benchmark **driver**kl specifies an IDE hard drive controller based on an operating system interface specification [188]. The parameter $k \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ encodes the level of manual abstraction that has been applied when translating the benchmark into a safety specification. The value a means that no abstraction has been applied, and the value d means that many details have been simplified. The parameter $l \in \{5, 6, 7, 8\}$ is a bound on the reaction time. The benchmark is only realizable for $l = 8$.

The remaining benchmarks are LTL formulas that are contained as examples in the distribution of the synthesis tool Acacia+ [40]. They have been translated into safety specifications using the approach by Filiot et al. [86]. The **demo**kl benchmarks represent LTL formulas that have originally been used as benchmarks for the synthesis tool Lily [125]. Here, k is just a running number without any special meaning and l is a bound for the liveness-to-safety transformation. Some of these benchmarks are unrealizable. The benchmark **gb**k represents a different formulation of the generalized buffer benchmark **genbuf** for two senders and two receivers. The parameter k is here a bound for the liveness-to-safety transformation. One of these instances is unrealizable. The benchmark **load**kl contains a specification of a load balancing system [81] for k clients that has been used as a case study for the Unbeast synthesis tool [81]. The parameter l is again a bound for the liveness-to-safety transformation. One of these instances is unrealizable. Finally, the benchmarks **ltl2dba**kl and **ltl2dpa**k from the Acacia+ [40] examples have been translated. Here, k is just a running index without any special meaning, and l is again a parameter of the translation. From these benchmarks, some are also unrealizable.

3.3.3 Strategy Computation Results

In this section, we compare different methods for strategy computation. Methods for computing circuits that implement a given strategy will be evaluated in Section 3.3.4. First, we will describe the compared methods and their configuration. Section 3.3.3.2 will then present performance results on the average over all our benchmarks. A more detailed investigation for the individual benchmark classes is then performed in Section 3.3.3.3. Section 3.3.3.4 will finally highlight other interesting observations. All experiments reported in this section were performed on an Intel Xeon E5430 CPU with 4 cores running at 2.66 GHz, and a 64 bit Linux.

3.3.3.1 Evaluated Configurations

Table 3.2 summarizes the methods and their configurations we compare in this thesis.

Baseline. BDD denotes a BDD-based implementation of the standard SafeWin procedure presented in Algorithm 2.1. It has been implemented by students and won a synthesis competition that

Table 3.2: Configurations for computing a winning strategy.

Name	Algorithm and Optimizations	Solver
BDD	SafeWin (Alg. 2.1)	CuDD
IFM	Re-implementation of [163]	MiniSat
ABS	AbsSynthe 2.0 [43]	CuDD
Q	QbfWin (Alg. 3.1)	DepQBF
QB	QbfWin (Alg. 3.1)	DepQBF + Bloqqer
QGB	QbfWin (Alg. 3.1) + Opt. RG (Sect. 3.1.4.1)	DepQBF + Bloqqer
QGAB	QGB + computing all generalizations (Sect. 3.1.1.3)	DepQBF + Bloqqer
QGCB	QbfWin (Alg. 3.1) + Opt. RG and RC (Sect. 3.1.4)	DepQBF + Bloqqer
QI	Incremental QbfWin with variable pool (Sect. 3.1.1.4)	Incremental DepQBF
S	SatWin1 (Alg. 3.4)	MiniSat
SG	SatWin1 (Alg. 3.4) + Opt. RG (Sect. 3.1.4.1)	MiniSat
SGC	SatWin1 (Alg. 3.4) + Opt. RG and RC (Sect. 3.1.4)	MiniSat
SE	SatWin1 (Alg. 3.4) + Expansion (Sect. 3.1.3)	MiniSat
SGE	SatWin1 (Alg. 3.4) + Opt. RG + Expansion	MiniSat
TQC	Eq. 3.3 + CNF Templates (Sect. 3.1.5.1)	DepQBF
TBC	Eq. 3.3 + CNF Templates (Sect. 3.1.5.1)	DepQBF + Bloqqer
TSC	Eq. 3.3 + CEGIS (Alg. 3.5) + CNF Templates	MiniSat
EPR	Reduction to EPR (Sect. 3.1.6.2)	iProver
P2	Parallel (Sect. 3.1.7) with 2 threads	MiniSat + DepQBF + Bloqqer
P3	Parallel (Sect. 3.1.7) with 3 threads	MiniSat + DepQBF + Bloqqer

has been carried out in a lecture. It is fairly optimized: it uses dynamic variable reordering, forced reorderings at certain points, combined BDD operations, and a cache to speed up the construction of the transition relation. See Section 2.3.1 for more background. IFM denotes a reimplementation of the approach by Morgenstern et al. [163]. It is inspired by the model checking algorithm IC3 [41] and based on SAT solving. AbsSynthe is a BDD-based synthesis tool that uses abstraction and refinement[10] as well as many other advanced optimizations [43]. It won the sequential synthesis track in the SyntComp 2014 [117] competition. In version 2.0 (the version we compare to), AbsSynthe has also been extended with an approach for compositional synthesis. AbsSynthe can therefore be considered as one of the leading state-of-the-art synthesis tools for safety specifications at the time of writing this thesis. Together with IFM and BDD, it serves as a baseline for our comparison. Since this section only evaluates the strategy computation, the circuit extraction is disabled in all baseline tools for now.

QBF-based learning. The configurations starting with a Q represent different realizations of the QbfWin procedure shown in Algorithm 3.1. This includes the basic algorithm with QBF preprocessing (QB) and without preprocessing (Q), a version (QGB) using optimization RG (see Section 3.1.4.1), and a version (QGCB) that also uses optimization RC (see Section 3.1.4.2). Furthermore, we present results for an implementation (QGAB) that computes all counterexample generalizations instead of just one (see Section 3.1.1.3), and for one of our three approaches (named QI) for incremental QBF solving (see Section 3.1.1.4). The results for the other two methods using incremental QBF solving are similar and can be found in the downloadable archive. The archive also contains other combinations of the different options and optimizations (20 in total).

Learning based on SAT solvers. The different configurations of the SAT solver based learning pro-

[10] Abstraction and refinement are applied (roughly) in the following way. Only a subset of the state variables are considered. Based on this subset, an under-approximation and an over-approximation of the mixed preimage operator Force_1^s are defined. These are used to compute an over-approximation $W\!\uparrow$ and an under-approximation $W\!\downarrow$ of the winning region. If the initial state is in $W\!\downarrow$, the specification is realizable. If it is not contained in $W\!\uparrow$, the specification is unrealizable. Otherwise, the abstraction is refined by considering additional state variables.

cedure SATWIN1, presented in Algorithm 3.4, are all named with an S as a first letter. Our comparison contains a plain implementation (S), a variant (SG) with optimization RG (see Section 3.1.4.1), and a version (SGC) that also performs optimization RC (see Section 3.1.4.2). The former two are also combined with our heuristic for performing universal expansion (see Section 3.1.3), named SE and SGE respectively. To simply the matters, we only present result using the SAT solver MiniSat. Results using Lingeling and PicoSAT can be found in the downloadable archive. Both Lingeling and PicoSAT can be faster than MiniSat for individual benchmark instances, but MiniSat yields better results on average.

Template-based approach. All configurations of our template-based approach (see Section 3.1.5) start with a T. A QBF-based implementation with and without QBF preprocessing is realized in TBC and TQC, respectively. TSC denotes a SAT solver based realization using our variant of the CEGIS algorithm (see Algorithm 3.5). We only present results using CNF templates. The results when using AND-inverter graph templates are similar and can be found in the downloadable archive.

Reduction to EPR. The configuration realizing the approach of Section 3.1.6.2 is named EPR.

Parallelization. The results produced by our parallelization with one thread are essentially the same as for SGE because our parallelization executes SGE when used with one thread. The additional communication overhead is negligible. The configurations with two and three threads are named P2 and P3, respectively. The additional speedup we achieve with more than three threads is rather insignificant. Thus, we do not present any results with more threads.

3.3.3.2 The Big Picture

We executed the configurations listed in Table 3.2 with a timeout of 10 000 seconds per benchmark instance and a memory limit of 8 GB. Figure 3.14 gives an overview of the resulting execution times in form of a cactus plot. The horizontal axis contains the benchmarks, sorted in the order of increasing execution times (individually for each configuration). The vertical axis shows the corresponding execution time on a logarithmic scale. Hence, the lines for the individual configurations can only rise, and the steeper a line rises, the worse is its scalability. Another way to read cactus plots is as follows: For a given time limit on the vertical axis, the horizontal axis contains the number of benchmarks that can be solved within this time limit. We omitted some of the more exotic configurations from Table 3.2 (namely Q, QGAB, QGCB, SGC, and SE) to keep the plot readable. In the following paragraphs, we will focus on the most important observations based on Figure 3.14. A more detailed comparison will be given in the next subsections.

Our reduction to EPR does not scale well. EPR could only solve 27 instances. In none of the cases, a timeout was hit. For all instances that could not be solved, iProver ran out of memory.

Our template-based configurations solve only few instances. By comparing the lines for TQC and TBC, we can see that QBF preprocessing improves the scalability of our QBF-based realization of the template-based approach quite significantly. Our implementation TSC using CEGIS and SAT solving can even solve a few more instances. In all three cases, the lines rise very steeply. Slightly oversimplified, this means that the template-based methods either find a solution quickly or not at all. Unfortunately, the latter case happens more often. In total, TSC solves only 115 instances, which is low compared to the other methods. However, the solved instances include some that cannot be solved by any other method, so the template-based approach can complement other techniques. We will elaborate on this aspect in the next section. Except for the (very large) gb benchmarks, the memory limit was never exceeded.

Incremental QBF solving gives a solid speedup for simple benchmark instances. Compared to QB and QGB, the realization QI using incremental QBF solving is faster on average by more than one order of magnitude for simple benchmark instances. For example, the 130 simplest instances for QB can all be solved by QB in less than 137 seconds each, while the 130 simplest instances for QI can be solved by QI in less than 6.2 seconds each. Yet, for more complex instances, QI falls behind QB. One possible reason is the lack of QBF preprocessing in QI, which appears to be a promising future research direction.

SAT solvers can outperform QBF solvers when learning a winning region. All our QBF-based

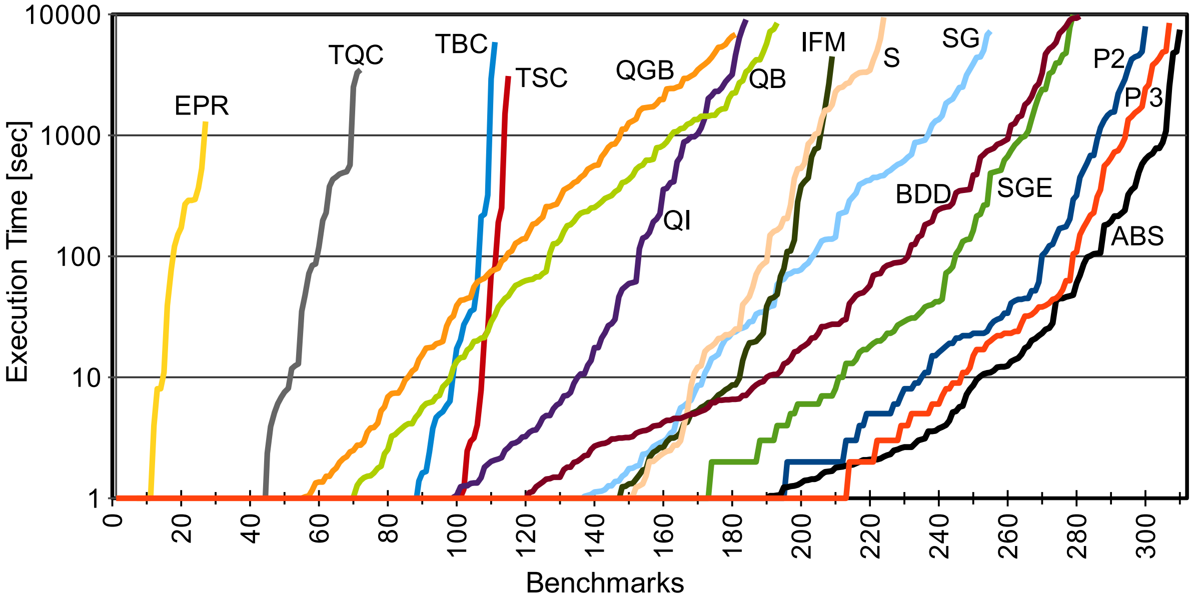


Figure 3.14: A cactus plot summarizing the execution times for computing a winning strategy with different methods and configurations. The vertical axis contains the execution time in seconds on a logarithmic scale. The horizontal axis gives the corresponding number of benchmarks that can be solved within this time limit. That is, the steeper a line rises, the worse is its scalability. The compared methods and configurations are summarized in Table 3.2. The configurations Q, QGAB, QGCB, SGC, and SE have been omitted to keep the plot readable.

methods are outperformed significantly even by the plain SAT solver based implementation S. The observation that it can be beneficial to solve QBF problems with plain SAT solving is not new [163, 120], and hence not completely surprising. The plain implementation S can already solve more instances than our reimplementation IFM of the approach by Morgenstern et al. [163].

Optimization RG yields a speedup of roughly one order of magnitude for method S. This can be observed, at least for larger benchmark instances, when comparing the lines for SG and S in Figure 3.14. For example, the 224 simplest instances for S can each be solved by S in at most 9450 seconds. On the other hand, SG can solve its 224 simplest instances in at most 470 seconds, which is 20 times shorter. Interestingly, optimization RG is *not* beneficial when applied to our QBF-based implementation (compare QGB versus QB) on the average over all our benchmarks, though. But even in the QBF case, it still yields a significant speedup for certain benchmark instances. While optimization RG turned out to be very effective, optimization RC does not have a positive effect on average in our experiments: the number of solved instances decreases from 255 to 236 when switching from SG to SGC (this is not shown in Figure 3.14 but can be seen in Table 3.3). But optimization RC is also beneficial for individual benchmark instances and, thus, not useless either.

Our heuristic for quantifier expansion gives a speedup of roughly one more order of magnitude. This can be seen by comparing the line for SGE with that for SG. Nailed down by numbers, SG solves its 254 simplest benchmarks in at most 6800 seconds each, while SGE solves its 254 simplest benchmarks in at most 268 seconds, which is 25 times shorter. The configuration SGE is already on a par with BDD.

Our parallelization achieves a speedup of more than one additional order of magnitude. SGE solves its 279 simplest benchmarks in at most 9920 seconds each. P2 solves its 279 simplest benchmarks in at most 295 seconds, which is 33 times shorter. P3 never requires more than 105 seconds on its 279 simplest benchmarks, which can even be seen as a speedup by a factor of 95 over SGE. Obviously, these speedups are not primarily the result of exploiting hardware parallelism. They rather stem from combining different approaches that complement each other. Although AbsSynthe uses advanced techniques such as abstraction/refinement, P3 is not far behind (P3 solves 4 instances less).

3.3.3.3 Performance per Benchmark Class

The previous section discussed the performance of the individual methods and configurations on average over all our benchmarks. In this section, we will perform a more fine-grained analysis for the different classes of benchmarks.

Table 3.3 lists the number of solved benchmark instances per benchmark class. Recall that a description of the compared methods and their configurations can be found in Table 3.2. Some statistics on the benchmarks can be found in Table 3.1. Also recall that all experiments were performed with a timeout of 10 000 seconds and a memory limit of 8 GB. Table 3.3 marks the "best" configuration for a certain benchmark class in blue: If several methods solve the same amount of instances, we marked the one with the lowest total execution time. For cases where the difference in the total execution time is insignificant, we marked several configurations. If most of the configurations solve all instances of a certain benchmark class in an insignificant amount of time, we refrain from marking them. Moreover, we do not include ABS and the parallelizations in this ranking because they combine several techniques.

add. Neither BDD nor the template-based method TBC require more than 0.2 seconds for any instance of the **add** benchmark. The SAT solver based learning approaches using universal expansion (SE, SGE) solve all instances as well, but require up to 42 seconds. Without expansion (S, SG, SGC), SATWIN1 requires many iterations to refine U before a counterexample is found or to conclude that no counterexample exists (see Algorithm 3.4). For instance, for **add**6y, roughly 4000 counterexample candidates are computed. This takes only one second. For **add**8y, SATWIN1 already computes 65 000 counterexample candidates, which takes 90 seconds. For **add**10y we hit the timeout. In contrast, the QBF-based learning methods (with names starting with Q) require only two iterations, but cannot solve significantly more instances either. This illustrates that the number of iterations alone is often not a good measure for estimating the performance of different algorithms relative to each other.

mult. The results for this benchmark are similar to **add**. The main difference is that the BDD-based implementation does not perform well, but this is not surprising since multipliers are known to be challenging for BDDs (see Section 2.3.1). Even ABS, which is highly optimized but also BDD-based, cannot solve all **mult** instances. The template-based configuration TBC performs best.

cnt. When the winning region is computed iteratively for this benchmark, this requires many iterations. More specifically, around 2^{k-1} refinements of the winning region are required for **cnt**ko. For $k = 30$, this already gives around half a billion iterations. Even though the time per iteration is very low for all configurations, this still results in timeouts for large values of k. In contrast, the template-based realizations require only one iteration. In particular, the configuration TSC solves all **cnt** instances in less than 8 seconds.

mv. Even though this benchmark has a relatively high number of inputs and control signals, most methods can solve all its instances within a fraction of a second. This benchmark will only be challenging for some of our circuit computation methods in Section 3.3.4.

bs. This benchmark contains a barrel shifter and is thus challenging for BDD. Most of the other methods solve all instances within a fraction of a second.

stay. This benchmark contains a counter and a multiplier, and thus combines the characteristics of **mult** and **cnt**. Hence, it is not surprising that one of the template-based configurations performs best.

amba and genbuf. While the previous benchmarks are basically toy examples designed to challenge the synthesis methods in different ways, the **amba** and **genbuf** benchmarks can be seen as specifications for realistic hardware designs. BDD performs very well on both these benchmarks. One circumstance contributing to this success may be that these benchmarks have been translated from input files for the BDD-based synthesis tool Ratsy [29], where they have been tweaked for efficient synthesizability. Yet, the SAT-based learning method SGE solves the same amount of **amba** instances as BDD, an is even slightly faster on the solved instances. For **genbuf**, BDD is unrivaled in our experiments.

fact and mov. None of our SAT-based methods can compete with BDDs on these benchmarks.

Table 3.3: Computing a winning strategy: solved instances per benchmark class. The timeout was set to 10 000 seconds. A memory limit was set to 8 GB. The first line gives the total number of benchmarks in the respective class. The last column gives the total number of benchmarks for which a winning strategy could be computed by the respective method within the given resource limits. A description of the strategy computation methods and their configurations is given in Table 3.2. The "best" (see the text for an explanation) configuration for each benchmark class is marked in blue.

	`add`	`mult`	`cnt`	`mv`	`bs`	`stay`	`amba`	`genbuf`	`fact`	`mov`	`driver`	`demo`	`gb`	`load`	`ltl2dba`	`ltl2dpa`	Total
Total	20	14	28	32	10	24	27	48	15	16	16	50	4	5	23	18	350
BDD	**20**	8	26	32	4	16	21	**48**	**12**	**11**	6	46	0	3	18	10	281
IFM	8	7	16	32	10	10	3	7	2	2	**16**	**50**	0	**5**	23	18	209
Q	8	4	24	30	10	10	2	7	1	0	2	44	0	2	20	15	179
QB	10	8	22	32	10	12	3	7	3	0	8	43	0	2	19	14	193
QGB	10	8	22	32	10	12	3	11	3	0	5	42	0	2	14	7	181
QGAB	10	8	22	32	10	12	3	11	3	0	4	42	0	1	6	9	173
QGCB	10	8	22	32	10	12	3	12	3	0	7	44	0	4	18	12	197
QI	8	4	22	30	10	10	3	7	2	0	2	45	0	3	22	16	184
S	8	7	24	32	10	18	15	13	2	2	3	46	0	4	23	17	224
SG	8	7	24	32	10	17	18	29	2	2	10	50	0	5	23	18	255
SGC	10	7	22	32	10	12	15	25	2	1	6	49	0	**5**	23	17	236
SE	20	9	24	32	10	12	19	20	3	2	3	48	0	4	23	18	247
SGE	20	9	24	32	10	12	**21**	37	5	3	10	**50**	0	**5**	**23**	**18**	279
TQC	10	9	24	12	4	10	0	0	0	0	0	3	0	0	0	0	72
TBC	**20**	**14**	24	25	6	18	0	0	0	0	0	4	0	0	0	0	111
TSC	8	7	**28**	32	10	**24**	0	0	0	0	0	6	0	0	0	0	115
EPR	4	2	11	8	0	2	0	0	0	0	0	0	0	0	0	0	27
P2	20	14	28	32	10	24	22	37	5	2	10	50	0	5	23	18	300
P3	20	14	28	32	10	24	23	37	5	2	16	50	0	5	23	18	307
ABS	20	9	24	32	10	16	27	48	11	11	7	50	0	5	23	18	311

driver. The IFM method by Morgenstern et al. [163] solves all instances of the **driver** benchmark in a fraction of a second. This is remarkable because with up to 326 state variables, these benchmarks are quite large. The SAT solver based learning methods SG and SGE are ranked second when run with optimization RG. Without optimization RG, only few instances can be solved.

demo. Both IFM and SGE can solve all instances in at most 40 seconds. With up to 280 state variables, the **demo** benchmarks contain quite large instances as well. The number of inputs is always relatively low, though.

gb. These benchmarks are far beyond reach with any of our methods. Even ABS fails.

load, ltl2dba and ltl2dpa. SGE performs best, solving most of these instances in less than a second. With 138 seconds, the longest execution time with SGE is also quite low.

Conclusions. Our QBF-based learning algorithms are dominated by our SAT solver based realizations across all benchmarks classes. EPR is even dominated by all other configurations. On the other hand, no single methods dominates all the other methods on all benchmark classes. We thus conclude that it is important to have different synthesis approaches available. Our experiments suggest that our novel SAT-based synthesis methods form an important contribution to the portfolio of available methods, complementing existing BDD-based methods (like BDD and ABS) but also existing SAT-based methods (like IFM).

3.3.3.4 Further Observations

This section highlights interesting observations that are more specific to certain methods.

QBF preprocessing is important. Figure 3.15 compares the execution times with and without QBF preprocessing for the QBF-based learning approach in a scatter plot. Each point in the diagram corresponds to one benchmark instance. The horizontal axis gives the execution time for the benchmark without preprocessing, and the vertical axis the corresponding execution time with preprocessing. Hence, all points below the diagonal represent a speedup due to preprocessing, and all points above are instances with a slowdown. Note that both axes are scaled logarithmically. We can see a slowdown by up to around one order of magnitude for many instances. However, there are also 20 points on the x-axis, indicating instances that can be solved in less than one second due to preprocessing. Furthermore, there are 19 points on the right border of the diagram, indicating cases where we had a timeout without preprocessing but get a solution when preprocessing is enabled. Two instances are even located in the lower right corner, representing an improvement from a timeout to less than one second. The number of solved instances increases from 179 to 193 due to preprocessing in the QBF-based learning method (see Table 3.3). The results for the template-based method (TQC versus TBC) are illustrated in Figure 3.16 and are even more impressive. A noticeable slowdown can only be observed for two cases. There are 44 points on the x-axis, for which preprocessing reduced the execution time to less than one second. For 40 cases, a timeout is avoided due to preprocessing. Finally, there are 23 points in the bottom right corner of Figure 3.16, for which a timeout is turned into a successful execution that takes less than one second.

Our optimizations for quantifier expansion can avoid a formula size explosion in many cases. For most of our SAT-based methods, the memory consumption is rather insignificant. As an exception, SGE can consume quite some memory due the expansion of universal quantifiers (see Section 3.1.3). However, our implementation can also fall back to SG if some memory limit is exceeded. In our experiments, this happened only for large instances of **mult**, **stay**, **gb**, and **driver**. One reason is our careful implementation of the expansion, which aggressively applies simplifications to reduce the formula blow-up. As an example, for **genbuf**15b, a straightforward implementation would produce $652 \cdot 2^{23} \approx 5 \cdot 10^9$ AND gates to define the expanded transition relation. With $3 \cdot 4 = 12$ byte per AND gate, this gives 56 GB. With our simplification techniques, the expanded transition relation has "only" 1.5 million AND gates. In addition, the final over-approximation F of the winning region W for **genbuf**15b contains 707 clauses and 8517 literals (after simplification). After negation, this makes 8517 clauses with 17739 literals. With 4 byte per literal, combining 2^{23} copies of this CNF in a straight-

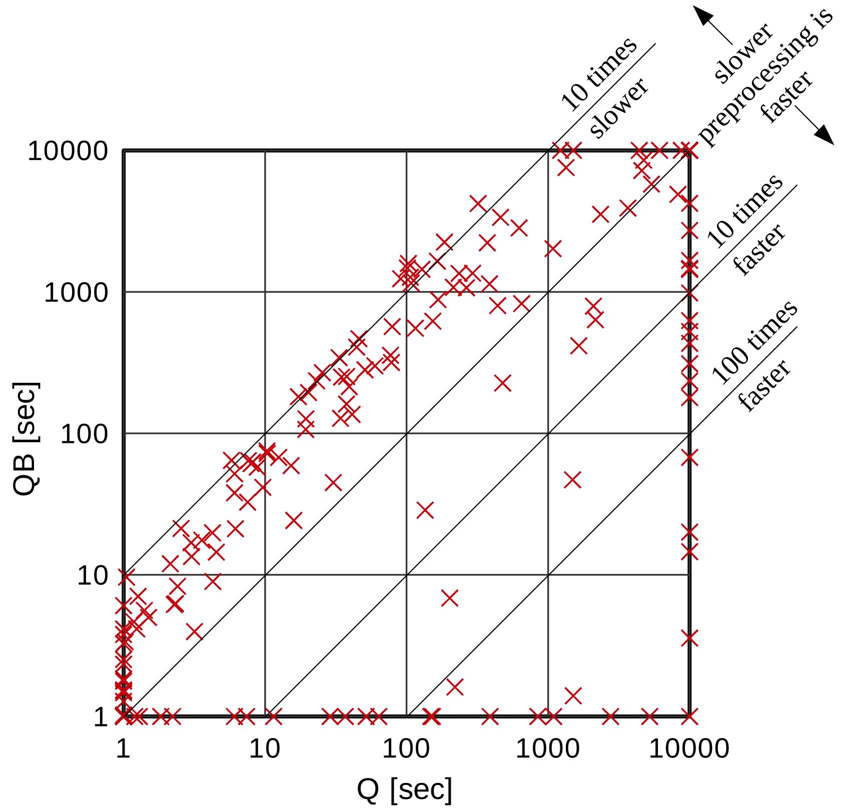


Figure 3.15: A scatter plot illustrating the speedup due to QBF preprocessing in QBF-based learning (Q versus QB). The x-axis contains the execution times without preprocessing, the y-axis with preprocessing. Note the logarithmic scale on both axes.

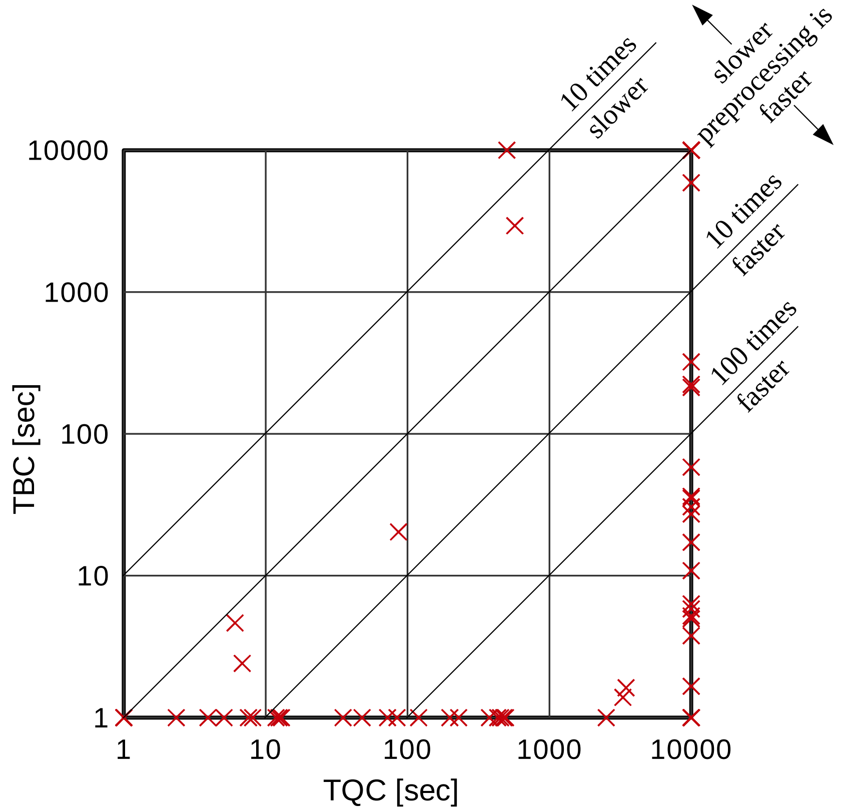


Figure 3.16: A scatter plot illustrating the speedup due to QBF preprocessing in our template-based approach (TQC versus TBC). The x-axis contains the execution times without preprocessing, the y-axis with preprocessing. Note the logarithmic scale on both axes.

forward way would require more than 500 GB of memory. Due to our simplifications, the expanded CNF for $\neg F$ has only 8.5 million literals and SGE solves this benchmark instance without falling back to SG. The maximum memory consumption is only 680 MB.

Our parallelization is more than a portfolio approach. When executed with two threads, our parallelization combines the template-based approach (in a mix of TBC and TSC) with the learning-based approach SGE. The two approaches do not only run in isolation, but share information: clauses discovered by the SGE-thread are communicated to the template-based thread and are considered as fixed part of the winning region there (see Section 3.1.7). This exchange of information can have a positive effect. For example, for the **genbuf** benchmark, the template-based approach fails to solve even the simplest instances when applied in isolation. However, in our parallelization, the final winning region for certain instances is actually found by the template-based thread. This includes even very large instances such as **genbuf**15b. The speedup of our parallelization P2 in comparison to SGE for such instances is rather moderate (e.g. $\approx 10\,\%$ for **genbuf**15b), but this still illustrates that complementary methods can also benefit from each other in our parallelization.

3.3.4 Circuit Synthesis Results

We now compare our different methods (from Section 3.2) to construct a circuit from a given strategy. Again, we first describe the evaluated configurations and the experimental setup. Section 3.3.4.3 then discusses the results on the average over all our benchmarks. Section 3.3.4.4 dives into more details by investigating the performance for different benchmark classes. Other interesting observations will be highlighted in Section 3.3.4.5.

3.3.4.1 Evaluated Configurations

Table 3.4 lists the different methods and their configurations compared in this section. All our SAT-based methods use the tool ABC [42] in a postprocessing step to further reduce the circuit size.[11]

Baseline. BDD denotes a BDD-based implementation of the standard cofactor-based approach presented in Algorithm 2.2. It is implemented in the tool that has already been discussed in Section 3.3.3.1, which won a synthesis competition that has been carried out in the course of a lecture. Besides dynamic variable reordering (with method SIFT [187, 198]), it also performs a forced reordering with a more expensive heuristic (SIFT_CONV [187, 198]) before circuits are extracted from the strategy. Furthermore, it uses a cache that maps BDD nodes to corresponding signals in the circuit constructed so far. Whenever new circuitry is added, the cache is consulted to reuse existing signals. Consequently, no two signals in the constructed circuit will be equivalent. The configuration ABS denotes the circuit synthesis step as implemented in AbsSynthe version 2.0 [43]. The basic algorithm is the same as that of BDD, but additional optimizations are applied. The IFM method by Morgenstern et al. [163], which has been used as a baseline in Section 3.3.3, is not included here because it can only compute a winning strategy but not a circuit implementing it.

QBF-based methods. Our approach using QBF certification (see Section 3.2.1) is named QC. A variant where we compute the negation of the winning region using the procedure NegLearn (Algorithm 3.6) is denoted by QCN. Our QBF-based learning approach from Algorithm 3.8 is used in three configurations: QL denotes a plain implementation using DepQBF, QLB also uses QBF preprocessing by Bloqqer, and QLI uses the DepQBF solver in an incremental fashion (see Section 3.2.2.2).

Interpolation-based method. An implementation of the interpolation-based method from Algorithm 3.10 is denoted by ID. It applies the dependency optimization presented in Section 3.2.3.2 and uses

[11] If the AIGER circuit has less than $2 \cdot 10^5$ AND gates before optimization, then we execute the command sequence `strash; refactor -zl; rewrite -zl;` three times, followed by `dfraig; rewrite -zl; dfraig;`. Between $2 \cdot 10^5$ and 10^6 AND gates, we only execute the sequence `strash; refactor -zl; rewrite -zl;` twice. For more than 10^6 AND gates, we perform it only once.

Table 3.4: Configurations for computing a circuit that implements a given strategy.

Name	Algorithm and Optimizations	Solver
BDD	CofSynt (Alg. 2.2)	CuDD
ABS	AbsSynthe 2.0 [43]	CuDD
QC	QBF Certification (Sect. 3.2.1)	QBFCert
QCN	QBF Certification (Sect. 3.2.1) + NegLearn (Alg. 3.6)	QBFCert
QL	SafeQbfSynt (Alg. 3.8)	DepQBF
QLB	SafeQbfSynt (Alg. 3.8)	DepQBF + Bloqqer
QLI	SafeQbfSynt (Alg. 3.8)	Incremental DepQBF
ID	SafeInterpolSynt (Alg. 3.10) + Dep. Opt. (Sect. 3.2.3.2)	MathSAT
SL	SafeInterpolSynt + CnfInterpol (Alg. 3.11)	Lingeling
SLD	SafeInterpolSynt + CnfInterpol (Alg. 3.11) + Dep. Opt. (Sect. 3.2.3.2)	Lingeling
SLDM	SafeInterpolSynt + CnfInterpol (Alg. 3.11) + Dep. Opt. + Minimizing the final solution (Sect. 3.2.4.2)	Lingeling
P2	Parallel (Sect. 3.2.5) with 2 threads	Lingeling + DepQBF
P3	Parallel (Sect. 3.2.5) with 3 threads	Lingeling + DepQBF

MathSAT version `5.2.12` as interpolation engine. MathSAT supports several interpolation methods. In our experiments we use McMillan's system [159]. Results with other interpolation methods are rather similar, though. We also implemented our own interpolation engine by processing PicoSAT proofs in the TraceCheck format[12]. However, for larger benchmark instances, the proof files grew prohibitively large with this approach. Our realization using MathSAT does not have this problem.

Learning based on SAT solvers. The configuration SL implements the SAT solver based learning approach from Section 3.2.4 without the dependency optimization (Section 3.2.3.2) and without minimizing the final solution (Section 3.2.4.2). SLD denotes a similar configuration, but with the dependency optimization enabled. Finally, the SLDM configuration also applies a minimization of the final solution by attempting to eliminate literals and clauses from the computed solutions (Section 3.2.4.2). All these configurations use activation variables to perform incremental solving across all calls to CnfInterpol (see Section 3.2.4.2). Lingeling is slightly faster on average than MiniSat and PicoSAT in all these configuration. Results for other configurations (28 in total) can be found in the downloadable archive.

3.3.4.2 Experimental Setup

Again, all experiments were performed on an Intel Xeon E5430 CPU with 4 cores running at 2.66 GHz, using a 64 bit Linux as operating system. A timeout was set to 10 000 seconds for all circuit synthesis runs. The available main memory was limited to 8 GB. The maximum size for auxiliary files to be written to the hard disk was set to 20 GB.

Sanity checks. All synthesized circuits were model checked using IC3 [41]. IC3 never found a counterexample but in some cases hit a timeout. We thus also ran a bounded model checker (BLIMC, which is distributed with Lingeling [23]) to get bounded correctness guarantees for such cases.

Winning strategies. For all our SAT-based circuit computation methods, we used the winning strategies as computed by configuration P3 (see Table 3.2). Preliminary experiments with other strategy computation methods suggest that the impact on the performance in circuit synthesis is rather small. One reason is that we simplify the computed winning region (or winning area) by calling CompressCnf (Algorithm 3.2) as a preprocessing step to circuit extraction (see Section 3.2). We thus refrain from running experiments with all combinations of our strategy- and circuit computation methods. Furthermore,

[12] `http://fmv.jku.at/tracecheck/` (last visit on 2015-08-01).

we stored the winning strategies computed by P3 into files and loaded them for our circuit computation experiments in order to ensure that all our methods operate on exactly the same strategy (and to save computational resources for recomputing the strategy each time). For BDD and ABS, we used the winning regions as computed by these tools as a starting point for circuit synthesis.

Benchmarks. From the 350 benchmark instances used to evaluate our strategy computation methods (see Section 3.3.2), only 267 instances have been used to compare our circuit computation methods. This has two reasons. First, 40 instances are unrealizable, so they have no winning strategy. Second, for some instances, no winning strategy could be computed (even with a timeout of 10^5 seconds) for at least one of the compared methods. In order to have a fair comparison, we thus used only those benchmarks for which P3, BDD and ABS succeeded in computing a winning strategy. In detail, P3 failed to compute a winning strategy for 25 instances. For 19 additional instances, BDD could not find a winning strategy within 10^5 seconds. This includes **cnt**$30\mathrm{n}$ and **cnt**$30\mathrm{y}$, for which we estimated BDD's circuit synthesis time (to be 0.1 seconds) and the circuit size (to be 32 gates) based on observations from smaller instances. This leaves 17 excluded instances. ABS could not compute a winning strategy for 11 more benchmarks. However, for two **cnt** instances, we could estimate the size and time to 0.1 seconds and 1 gate based on results for smaller instances. For eight instances of the **stay**ko benchmark, we also estimated 0.1 seconds and $k+1$ gates. What remains is one **driver** instance to exclude. This finally results in $350 - 40 - 25 - 17 - 1 = 267$ instances used for the comparison. The number of used instances per benchmark class can be seen from Table 3.5 (see the line labeled by "Total").

More detailed comparisons. The downloadable archive also contains more detailed pairwise comparisons on larger subsets of the benchmark instances. This includes charts to compare our SAT-based methods with ABS on all 281 benchmarks for which both ABS and P3 were able to compute a winning strategy. Charts comparing our SAT-based methods with BDD on all 268 instance on which both BDD and P3 were able could find a winning region are included as well. However, since the results are almost identical to our three-way comparison, we refrain from presenting them in this document.

3.3.4.3 The Big Picture

The Figures 3.17 and 3.18 contain cactus plots illustrating the execution time and the resulting circuit size for the method configurations from Table 3.4. The configuration QC has been omitted in the plots because both the execution time and the circuit size is similar to QCN (the difference is mostly in the memory consumption). The configuration QL performs slightly worse than QLB and QLI and has also been omitted to make the plots more legible. The following paragraphs discuss the most important observations based on these two figures. A more detailed analysis will be done in Section 3.3.4.4 and 3.3.4.5.

QBF certification does not perform well. From Figure 3.18, we can see that the QBF certification method QCN produces the largest circuits. Figure 3.17 illustrates that QCN is on average slightly faster than QLB, but still solves less instances within the given resource limits. The reason is that QCN often exceeds the 20 GB limit for auxiliary files because the proof traces produced by DepQBF in the QBFCert framework can grow very large (several hundreds of GB when run without limits).

QBF-based learning is slow but produces small circuits. Especially when used with incremental QBF solving, QBF-based learning can outperform QBF certification both regarding execution time and circuit size (compare QLI versus QCN). Still, in comparison with the other methods, QLI is on average way slower. Regarding circuit size, QLI and QLB are on a par and outperform the interpolation-based method ID as well as the BDD-based implementation BDD by almost one order of magnitude on average.

The interpolation-based approach does not outperform BDDs in our experiments. Regarding circuit size, the interpolation-based configuration ID yields similar results as BDD on average. However, ID is noticeably slower than BDD, especially for more complex benchmark instances.

Our SAT solver based learning approach outperforms all our other non-parallel methods. This holds true for both the execution time and the circuit size. The configuration SLD turns out to be our best non-parallel option on the average over all our benchmarks. It is already faster than BDD on average by

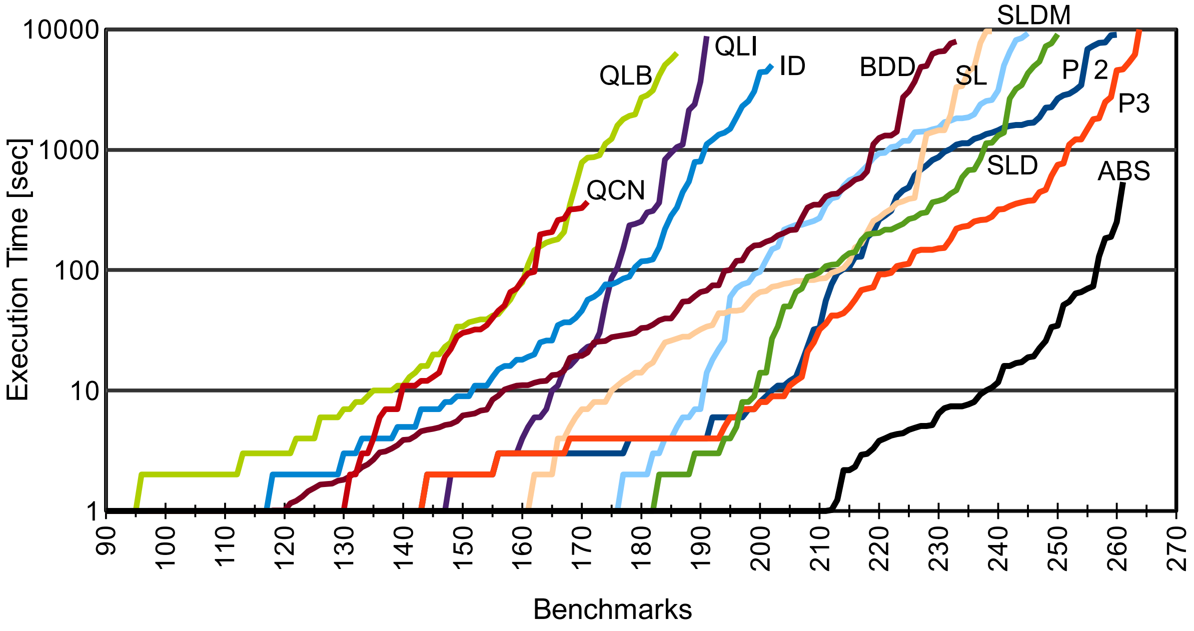


Figure 3.17: A cactus plot summarizing the execution times for computing a circuit that implements a given winning strategy with different methods and configurations. The y-axis contains the execution time in seconds on a logarithmic scale. The x-axis gives the number of benchmarks that can be solved within this time limit. The compared configurations are summarized in Table 3.4.

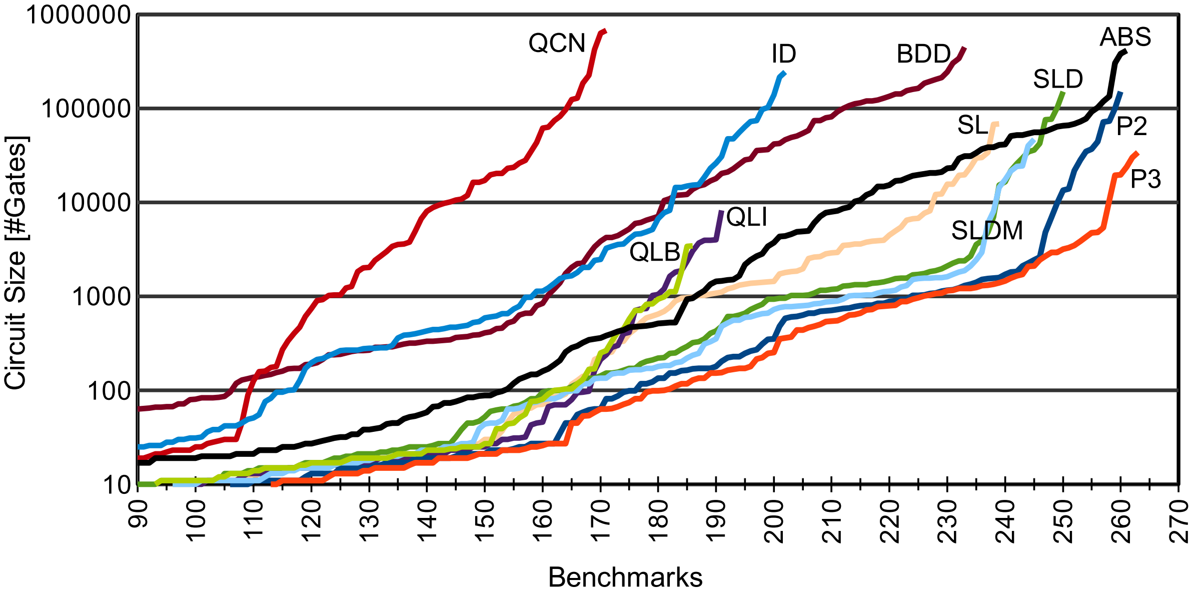


Figure 3.18: A cactus plot summarizing the resulting circuit sizes for our different methods to compute a circuit from a given strategy. The y-axis shows the number of AND gates in the AIGER circuit that defines the control signals. The x-axis gives the number of benchmarks that can be solved within this size limit. The compared configurations are summarized in Table 3.4.

more than one order of magnitude. For example, BDD can solve its 233 simplest benchmark instances in at most 7931 seconds each. SLD never needs more than 461 seconds for its 233 simplest instances. This is a factor of 17. With respect to circuit size, the situation is even more extreme. The 233 smallest circuits produced by BDD have at most half a million AND gates each. The 233 smallest circuits produced by SLD have at most 2383 gates each. This is smaller by a factor of 188. SLDM produces even slightly smaller circuits, with an improvement factor of 240 for the 233 smallest circuits compared to BDD. This is not only because BDD produces large circuits for benchmarks that cannot be solved by SLD. The most extreme instance is **driver**d8[13], for which BDD produces a circuit with 3.7 million AND gates, while SLD produces a circuit with only 66 gates.

Our parallelization is competitive with the state of the art. Our parallelization P3 increases the number of solved instances compared to SLD from 250 to 263 by combining different methods and optimizations. The circuit sizes also decrease slightly, which is partly due to our heuristics performing additional circuit minimizations if there is sufficient time left (see Section 3.2.5), and due to selecting the smallest solution from all threads. Our parallelization already solves 2 instances more than the state-of-the-art tool AbsSynthe. When comparing P3 against AbsSynthe in Figure 3.17, we can see that AbsSynthe solves many instances in less than one second but the execution times grow steeper for more difficult instances. Thus, AbsSynthe can be superior if the timeout is short, while P3 shows a more steady pace. Regarding circuit size, our parallelization outperforms AbsSynthe by more than one order of magnitude on average (compare P3 versus ABS in Figure 3.18).

Execution time and circuit size often correlate. With the exception of the QBF-based learning methods, we can observe a correlation between execution time and circuit size in our experiments. Methods that are fast have a tendency to also producing small circuits and vice versa. At the first glance, this may be surprising because, intuitively, one could expect that we have to find a good trade-off between these performance metrics. One reason for the correlation is that most of our methods (all except QC and QCN) compute circuits iteratively for one control signal after the other. After every iteration, the strategy formula is refined with the solutions for the control signals that have been synthesized so far. If these solutions are complicated, then this results in more complicated strategy formulas for the next iterations, which can increase the computation times. For the learning-based methods, the size of the CNF formulas defining the control signals directly corresponds to the number of iterations that were needed to compute them: Every clause results from a mayor iteration involving a counterexample computation. Every literal in a clause witnesses a failed attempt to eliminate this literal with a SAT- or QBF solver call. A correlation between the circuit size and the execution time is thus natural.

Computing circuits from strategies is by no means a negligible step in the synthesis process. Let us compare the total strategy computation time against the total circuit computation time for all instances where both steps terminate within the timeout of 10 000 seconds. For P3, this comparison reveals that 52 percent of the total synthesis time is spent on strategy computation and 48 percent is consumed by circuit computation. For BDD, the distribution is 60 % to 40 %. Only for ABS, the distribution is 90 % to 10 %, which may be due to the abstraction/refinement techniques implemented in ABS.

3.3.4.4 Performance per Benchmark Class

This section analyzes the performance of our methods for circuit synthesis for the different benchmark classes. We will see that the configuration SLD is not always superior.

Table 3.5 lists the number of benchmark instances that could be solved per benchmark class by the different configurations. The fastest configuration is marked in blue. If the same amount of instances are solved by several configurations, we marked the one with the lowest total execution time. In case the total execution time is very similar, we sometimes marked several configurations. For benchmark classes where most of the configurations solve all instances, we did not mark any configuration. Again, we do not include ABS and the parallelizations in this ranking because they combine several techniques.

[13] This instance is not included in Figure 3.18 because ABS could not compute a winning strategy for this instance.

Table 3.5: Computing a circuit from a strategy: solved instances per benchmark class. The timeout was set to 10 000 seconds. A memory limit was set to 8 GB. The maximum size of auxiliary files was limited to 20 GB. The first line gives the total number of benchmark instances in the respective class. The last column gives the total number of instances for which a circuit could be computed by the respective method within the given resource limits. A description of the circuit computation methods and their configurations is given in Table 3.4. The "fastest" (see the text for an explanation) configuration for each benchmark class is marked in blue.

	`add`	`mult`	`cnt`	`mv`	`bs`	`stay`	`amba`	`genbuf`	`fact`	`mov`	`driver`	`demo`	`load`	`ltl2dba`	`ltl2dpa`	Total
Total	20	14	28	32	10	24	23	42	3	2	0	37	3	19	10	267
BDD	**20**	7	28	32	4	16	13	41	**3**	**2**	-	36	3	18	10	233
QC	6	4	28	32	10	24	0	6	2	0	-	30	2	7	10	161
QCN	6	4	28	32	10	24	3	10	3	1	-	31	2	9	8	171
QL	8	4	28	32	10	24	3	10	2	0	-	35	3	19	10	188
QLB	8	3	28	32	10	24	3	9	2	0	-	35	3	19	10	186
QLI	8	4	28	32	10	24	3	11	3	1	-	35	3	19	10	191
ID	20	5	28	28	10	24	4	12	2	1	-	36	3	**19**	**10**	202
SL	12	5	28	25	10	24	**20**	**41**	**3**	2	-	**37**	**3**	**19**	**10**	239
SLD	**20**	**14**	28	22	10	24	17	41	3	2	-	37	**3**	**19**	**10**	250
SLDM	**20**	**14**	28	22	10	24	14	40	3	1	-	37	**3**	**19**	**10**	245
P2	20	14	28	32	10	24	17	41	3	2	-	37	3	19	10	260
P3	20	14	28	32	10	24	20	41	3	2	-	37	3	19	10	263
ABS	20	8	28	32	10	24	23	42	3	2	-	37	3	19	10	261

add. The configurations BDD, SLD and SLDM solve all instances of the **add** benchmark within one second. The difference in circuit size is moderate (at most 171 gates with SLD and SLDM; at most 416 gates with BDD). The interpolation-based method ID solves all instances as well but requires at most 40 seconds. The good results of SLD, SLDM and ID are mostly due to our dependency optimization (see Section 3.2.3.2 and Section 3.2.4.2): Without the dependency optimization, the SAT solver based learning method solves only 12 instances (SLD versus SL).

mult. This benchmark is similar in spirit as **add**, but the circuit to be synthesized is more complex. SLD and SLDM still perform well, again due to dependency optimization. However, BDD and ID fall back noticeably. The difference in circuit size also grows more significant: For example, BDD implements **mult**9 with more than 10^5 gates, while SLD and SLDM require only 633 gates. One reason is that multipliers cannot be represented by small (monolithic) BDDs with any variable ordering (see Section 2.3.1). Since the BDD method dumps BDDs as a network of multiplexers to obtain the resulting circuit, the BDD size does not only affect the computation time but also the resulting circuit size. Our SAT solving based methods SLD and SLDM do not suffer from this issue. They even outperform AbsSynthe significantly on this benchmark.

cnt, bs and stay. These benchmarks can be solved by all our methods in a few seconds. Only ID requires up to 46 seconds on larger instances of **stay**. BDD performs well on **cnt**, but cannot solve all instances of **bs** and **stay**. The latter two benchmarks contain barrel shifters and multipliers, which are known to be challenging for BDDs.

mv. The **mv** benchmark is an interesting case. Most of our methods can solve all instances of this benchmark in less than one second. However, for the interpolation-based method ID as well as the SAT solver based learning methods SL, SLD and SLDM, this benchmark is challenging. All these methods are based on InterpolSynt (Algorithm 3.9). The crux with the **mv** benchmark is that the XOR sum of all control signals must be true. InterpolSynt starts by building a circuit to fix the value of the last control signal based on all other control signals such that this is ensured. Since this circuit needs to react properly to all possible values of all other control signals, it can be very large. In particular, the SAT solver based learning methods build this circuit in a CNF representation without introducing auxiliary variables. A CNF formula that computes the XOR sum of n variables without introducing new auxiliary variables requires 2^{n-1} clauses. For **mv**28y, this gives $2^{27} \approx 134 \cdot 10^6$ clauses.[14] Only in the last iteration, when the algorithm processes the first control signal, InterpolSynt discovers that this signal can actually be set to a constant value. This has the effect that all the computed circuits for the other control signals also collapse to constant values. The root cause for this behavior is that InterpolSynt is very conservative with exploiting implementation freedom (see Section 3.2.3.4 for a discussion). In contrast, the QBF-based learning algorithm QbfSynt (Algorithm 3.7) exploits the available freedom greedily. It sets each control signal to a constant value right away, because this is sufficient to ensure that a solution for the remaining control signals still exists.

amba and genbuf. For these benchmarks, the SAT solver based learning configuration SL performs best. That is, the dependency optimization implemented in SLD and SLDM does not pay off. SLD, SL and BDD can solve the same amount of **genbuf** instances, but SLD is slower than SL by a factor of 2 in total, and BDD is even slightly slower than SLD in total. The sum of the circuit sizes for all **genbuf** instances is 44 times smaller when using SL instead of BDD. For **amba**, the factor is 21 when counting only the instances that can be solved by both SL and BDD.

fact and mov. Both BDD and SL can solve all **fact** instances in less than 10 seconds per instance. The **mov** instances are solved by BDD in at most 160 seconds per instance. The second fastest configuration for **mov** is SL, but it requires already 4500 seconds.

driver. ABS cannot compute a winning strategy for any of the **driver** instances, so this benchmark is not included in the comparison of Table 3.5. Our SAT solver based learning methods SL, SLD and SLDM can solve all **driver** instances in less than 10 seconds. The circuit size with these methods

[14] For the resubstitution step in Line 9 of Algorithm 3.9, this CNF also needs to be negated, which can even result in running out of memory.

is at most 600 gates. BDD can only handle the smallest **driver** instance, but takes already half an hour to produce a circuit with 3.7 million gates. With up to 326 state variables and 98 inputs, the **driver** benchmark certainly offers plenty of possibilities for building complicated circuitry. Yet, in contrast to BDD, our learning-based methods appear to perform well in exploiting the implementation freedom to avoid overly complicated solutions.

demo. Only ABS and our SAT solver based learning methods SL, SLD and SLDM can solve all instances. The dependency optimization is not beneficial: SLD is slower than SL by a factor of 3.2.

load. Again, the SAT solver based learning methods SL, SLD and SLDM perform best: they solve all instances in at most 2 seconds. ID requires up to 18 seconds. The fastest QBF-based learning method is QLI, requiring up to 87 seconds for the **load** instances. BDD requires up to 10 minutes.

ltl2dba and ltl2dpa. The configurations ID, SL, SLD, and SLDM require at most 4 seconds on these benchmarks. Other configurations that can also solve all these benchmarks are slightly slower.

3.3.4.5 Further Observations

This section highlights some other, interesting observations.

The effect of our postprocessing with ABC [42] is rather insignificant. For SLD, ABC manages to reduce the average circuit size from 9500 gates to around 2700 gates in our experiments. However, this average is strongly influenced by the **mv** benchmark, where circuits with up to half a million gates are reduced to circuits were all control signals are driven by constants. See Section 3.3.4.4 for an explanation why this happens. This reduction for the **mv** benchmark could also be achieved with a simple constant propagation. When omitting the **mv** benchmark, the average circuit size is reduced from 4100 gates to 2900 gates, which is a reduction by around 30 percent. In relation to the circuit size differences between our methods, which can be in the range of several orders of magnitude (see Figure 3.18), this is rather insignificant. On the other hand, in the case of SLD, only 0.6 percent of the total execution time for all benchmarks is spent by ABC. By modifying the sequence of minimization commands executed by ABC, other trade-offs between the execution time and the resulting circuit size improvements are possible. Yet, our experiments suggest that postprocessing cannot easily compensate the large circuit size differences between the methods. In other words, exploiting the implementation freedom cleverly while computing the circuits appears to be much more effective than investing more effort into postprocessing.

Incremental QBF solving outperforms QBF preprocessing in our circuit synthesis experiments. Figure 3.19 illustrates the effect of QBF preprocessing in our QBF-based learning method for circuit synthesis by comparing QLB against QL in a scatter plot. We can see a negative effect for most instances. The number of solved instances even decreases from 188 to 186 (see Table 3.5). By trend, preprocessing is more beneficial for more complex instances. Some of the more complex instances have been left out in the comparison because either BDD or ABS failed to compute a winning region. If we consider all 285 instances on which P3 managed to compute a winning strategy, the number of solved instances actually increases from 190 to 193 due to QBF preprocessing. Hence, preprocessing also has its merits. On the other hand, incremental QBF solving has an exclusively positive impact in our experiments. It is visualized in Figure 3.20, comparing QLI against QL. There is not a single instance where incremental QBF solving increased the computation time. In 3 cases, a timeout is avoided. When counting only the instances where QL terminates successfully, the average execution time is reduced from 204 to 59 seconds, which corresponds to a speedup factor of 3.5.

Using NEGLEARN reduces the memory required by QBFCert. As already mentioned, QBFCert can consume quite some memory. This applies to both main memory as well as disk space for auxiliary files. As a consequence, QC encounters a timeout for only two benchmark instances. For the other instances that cannot be solved, the reason is in exceeding the memory limit. When using NEGLEARN (Algorithm 3.6) in order to compute the negation of the winning region without introducing auxiliary variables, the size of the auxiliary files is reduced by up to a factor of 30. On the other hand, for more than 15 instances, running NEGLEARN only trades a memory issue for a timeout. Still, the total number

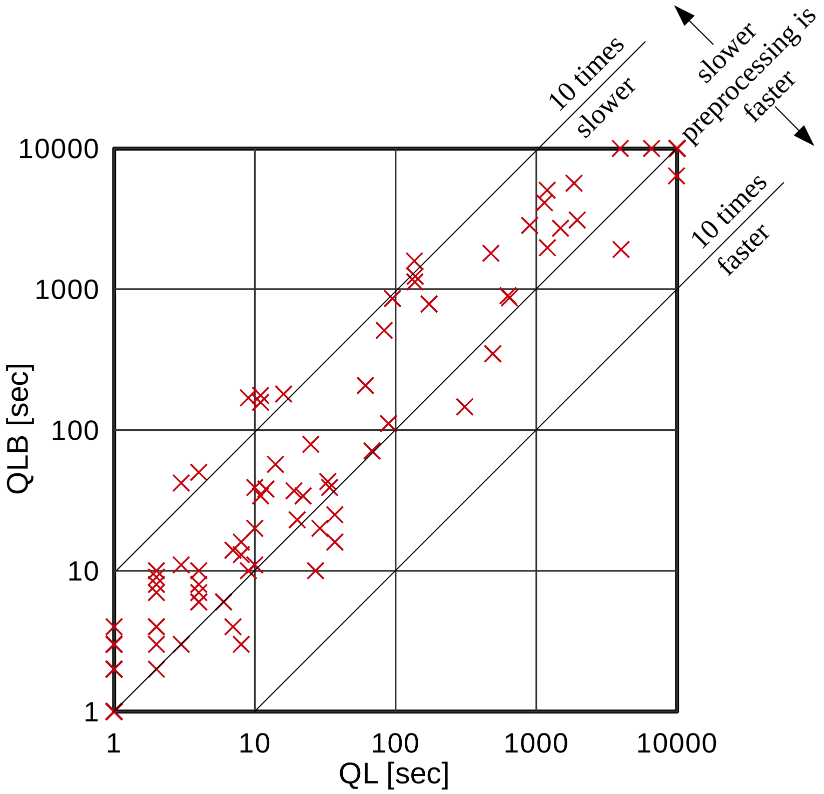


Figure 3.19: The effect of QBF preprocessing in circuit synthesis, illustrated in a scatter plot. The x-axis shows the execution times without preprocessing (configuration QL). The y-axis gives the corresponding execution times with QBF preprocessing enabled (configuration QLB). Note that both axes are scaled logarithmically.

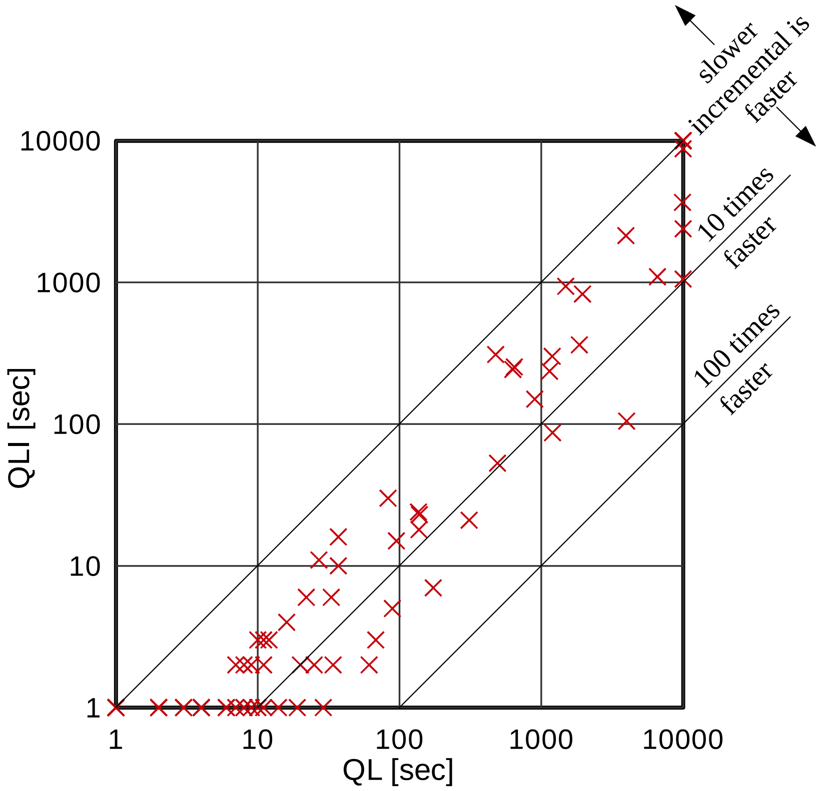


Figure 3.20: The effect of incremental solving in circuit synthesis, illustrated in a scatter plot. The x-axis shows the execution times without incremental QBF solving (configuration QL), the y-axis with incremental QBF solving enabled (configuration QLI).

of solved instances increases from 161 to 171 in our QBF certification approach (compare QC with QCN in Table 3.5). For our other methods, running NegLearn does not pay off, though.

Minimizing the final solution in SAT solver based learning yields moderate circuit reductions. When counting only the benchmark instances where both SLD and SLDM terminate, the average circuit size is reduced by 33% (from 1500 to 1000 gates) due to the final minimization step discussed in Section 3.2.4.2. On the other hand, the average circuit computation time increases from 106 seconds to 311 seconds, which is almost a factor of 3. For individual benchmark instances, the cost/benefit ratio can be lower, though. For example, in the case of **driver**b8, the circuit size is reduced from 594 gates to 152 gates in only a few extra seconds.

3.3.5 Discussion

Binary Decision Diagrams are increasingly displaced by SAT-based methods in the formal verification of hardware circuits. In synthesis, however, outperforming BDDs is challenging.

Outperforming BDDs in strategy computation. For the strategy computation step, our SAT solver based learning approach is competitive with the BDD-based reference implementation in our experiments, but only when making clever use of incremental solving, unsatisfiable cores, our optimization for exploiting unreachable states, and our heuristic for expanding quantifiers. With our parallelization, we can even solve significantly more benchmark instances than the BDD-based implementation. This is achieved by complementing the SAT solver based learning approach with our template-based approach. An advantage of BDDs is that they can handle both universal and existential quantification. This also holds true for QBF solvers. However, a QBF solver only computes one satisfying assignment, while BDDs eliminate the quantifiers to represent all satisfying assignments simultaneously. Our QBF-based algorithms have to compensate for that with more iterations. The performance of our QBF-based algorithms is rather limited compared to our SAT solver based realizations. This is somewhat surprising because the lack of universal quantification induces even more iterations. However, considering that QBF is still a rather young research discipline, this situation may change in the future. A combination of incremental QBF solving with preprocessing appears to be a particularly promising direction.
Our parallelization is on a par with the state-of-the-art synthesis tool AbsSynthe, which is also BDD-based but uses abstraction/refinement and other advanced optimizations. Adopting optimizations like abstraction/refinement from AbsSynthe appears to be a promising direction for future work.

Outperforming BDDs in circuit computation. For the second synthesis step, where circuits implementing the computed strategies are constructed, our satisfiability-based methods are even more beneficial in the average over all our benchmarks. In particular, our combination of interpolation with SAT solver based learning outperforms the BDD-based reference implementation by roughly one order of magnitude on average. Moreover, it produces circuits that are smaller by around two orders of magnitude on average. One reason is that the learning techniques we apply seem to be good at exploiting implementation freedom. In earlier work [82], we showed that learning techniques can also improve the circuit sizes when used with BDDs, but only at the cost of additional computation time. The experiments reported in this thesis suggest that the combination of learning algorithms with decision procedures for satisfiability is more promising. Our plain SAT solver based learning approach is still significantly slower than AbsSynthe. However, our parallelization can already solve more instances by combining different SAT-based methods. Moreover, it produces circuits that are smaller than those from AbsSynthe by more than one order of magnitude on average in our experiments.

Conclusion. BDDs are far from obsolete in the synthesis of reactive hardware systems. Yet, SAT-based methods can also be competitive, and even outperform BDD-based implementations on the average when designed carefully. We also observed that satisfiability-based methods can solve classes of benchmarks that are hard to deal with for BDD. We therefore believe that our novel satisfiability-based synthesis methods form an important contribution to the portfolio of available approaches.

4 SAT-Based Software Controller Synthesis for Program Repair

Parts of this chapter are based on my previous publications [140, 141, 145, 30].
References to these sources are not always made explicit.

The previous chapter discussed methods for hardware controller synthesis, where all inputs, control signals and state variables are represented by Boolean variables. The focus in this setting was on improving scalability with respect to the state of the art. To this end, we proposed novel algorithms based on templates or counterexample-guided refinements of solution candidates, and realized them using decision procedures for the satisfiability of quantified and unquantified propositional formulas.

In this chapter, we will present a controller synthesis approach for software programs and apply it in the context of automatic program repair. Our synthesis approach follows some of the same principles, namely templates to fix the structure of the solution and counterexample-guided refinement of solution candidates to find a proper template instantiation. However, program variables are no longer Boolean in the software setting but represented as domain variables in some theory. Consequently, we will use an SMT solver to reason about the program behavior formally. Besides scalability, this section also aims at applicability, in particular in the context of automatic program repair. Hence, we will not only address the synthesis of repairs but also program analysis and automatic fault localization in order to realize a fully automatic debugging flow. In order to produce meaningful diagnostic information for the user, our goal is the synthesis of fine-grained and readable repairs.

Outline. Our debugging flow is outlined in Figure 4.1. Its objectives will be elaborated in Section 4.1. The input, discussed in Section 4.1.1, consists of an incorrect software program P and a specification φ in the form of assertions in the code. Assertions can also be used to compare results computed by the program against that of a reference implementation. An extension to specifications that are given in the form of test cases will be presented in Section 4.6. Our debugging flow then consists of three steps. First, we apply program analysis to lift the debugging problem into the domain of logic by capturing correctness aspects of the program using formulas. Our approach for constructing these formulas using symbolic or concolic execution will be discussed in Section 4.2. Second, we perform automatic fault localization in order to identify parts of the program that could potentially be responsible for the incorrectness. Section 4.3 discusses a fault localization approach based on SMT solving, and Section 4.4 will present an alternative solution based on deductive verification and first-order theorem proving. The third step is to synthesize replacements for the potentially faulty program parts. A basic solution using templates and Counterexample-Guided Inductive Synthesis (CEGIS) will be given in Section 4.5. Section 4.7 will then propose a more advanced approach that interleaves the repair synthesis with on-the-fly program analysis using concolic execution. Finally, Section 4.9 will present our proof-of-concept implementation as well as experimental results.

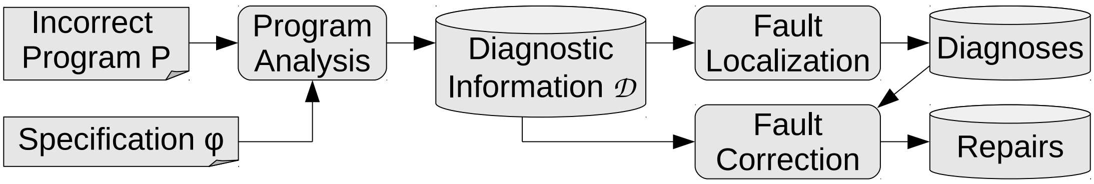


Figure 4.1: Outline of our automatic debugging flow. The input is an incorrect software program P and a specification φ. The output is a set of potential fault locations (diagnoses) and a set of suggestions how to fix the program (repairs). Our flow consists of three steps: program analysis, fault localization, and fault correction.

4.1 Objectives

In this section, we first discuss the input to our debugging flow, which consists of an incorrect program P and a specification φ. Based on the input definition, we then discuss objectives regarding the output of our automatic debugging approach as well as the envisioned usage scenario.

4.1.1 Input

The input of our debugging flow consists of an incorrect program P and a specification φ. We refrain from defining a formal syntax and semantics for programs because our debugging flow abstracts from the program in the first step already. The heart of our debugging flow then operates on formulas that capture only the relevant correctness properties of the program.

Programming language. We impose several assumptions about the program under analysis. Some of these assumptions are only made to simplify the presentation and can be relaxed. We consider programs written in an imperative programming language. We allow for both global and local program variables but assume that all variables store integers[1] of the same (fixed and finite) bit-width. That is, we do not consider floating point variables, pointers, arrays, compound types or type casts. All the usual arithmetic operations on integers (+, −, `*`, /, `%`), including bit-wise operations (`&`, `|`, `~`, `^`, <<, >>), are allowed. Furthermore, we support the usual comparison operators (==, !=, <, <=, >, >=). Besides assignments we also support the typical control flow primitives (`if-else`, `switch`, `while`, `do-while`, `for`) as well as function and procedure calls, which can also be recursive. Our proof-of-concept debugging tool operates on simple C programs and relaxes some of these restrictions. Section 4.9.1 will discuss which features of the programming language C are supported by our tool to which extent.

Program inputs. The program P can contain calls to a special function `input`, which takes no parameters and returns an unknown integer value, which is considered as an input to the program. Furthermore, all parameters of the entry function are implicitly considered to be inputs of the program.

Assertions and assumptions. Assertions are program statements of the form `assert(c)`, where `c` is a condition. Intuitively, an assertion expresses that the condition `c` must be true at that point in the program. Likewise, assumptions are statements of the form `assume(c)`, where `c` is a condition that is assumed to be true at that point in the program. We use assertions and assumptions as specification.

Specifications. Our basic approach takes the assertions in the code as a specification φ. We say that a program P *violates* its specification φ if there exist values for the inputs such that some `assert(c)` statement is executed with `c` being false and without executing an `assume(d)` statement for which `d` is false beforehand. Otherwise, P *satisfies* φ. We furthermore call a program *correct* if it satisfies its specification and *incorrect* otherwise. In Section 4.6, we will later extend our debugging flow to also support test cases as a specification. Since the problem of deciding whether a given program is correct is undecidable in general (see Section 2.8.1), our approach will only detect and repair correctness issues with certain approximations.

Incomplete specifications. We do not require specifications to be complete in the sense that they rule out every undesirable behavior. As a consequence, it can happen that our approach reports a repair as "correct" according to the definition above, but the repair may still not satisfy the design intent of the user. In this case, we suggest that the user refines the specification (with additional assertions) to rule out the undesirable repairs and starts the tool again. This way of keeping the user in the loop has the positive side-effect that the specification is improved during the debugging process as well.

[1] When we use the term "*integer*" in this section, we always refer to an integral data type ranging over a subset of all integer numbers that can be represented with some fixed and finite number of bits. We will use the term "*mathematical integer*" when referring to the (infinite) set of all integers.

4.1.2 Output and Objectives

The output of our debugging flow consists of diagnoses and repairs. In order to convey meaningful feedback to the user, we formulate additional objectives for this output.

Diagnoses and repairs. In a fist step, our debugging flow produces a set of diagnoses, which are program parts that could be responsible for specification violations. The diagnoses thus represent potential fault locations. Second, our flow also produces repairs, which are modified versions of the original program P that satisfy the specification (modulo approximations of the program semantics made during the repair computation). Repairs will be computed from diagnoses. That is, a repaired program will only differ from the original program P in program parts identified by a diagnosis.

Fine-grained diagnoses and repairs. As an additional objective, we want diagnoses to be fine-grained, i.e., to consist of relatively small program parts. Coarse-grained diagnoses, e.g., identifying functions of the program as potentially faulty, would leave a significant portion of the work in tracking down the fault to the user. Moreover, since our approach will infer repairs from diagnoses, fine-grained diagnoses will also result in fine-grained repairs that modify only small program parts. This has several advantages. First, the user must only comprehend small code changes in order to understand the repair. A completely resynthesized program function, for instance, can be much harder to understand. Second, small code changes have a higher probability to be in line with the user's design intent (recall that we do not require the specification to be complete). Third, the restriction to small code changes also restricts the search space for repairs and thus supports efficiency.

Readable repairs. Even fine-grained repairs can be difficult to understand for the user. For instance, a single `if`-condition can be repaired with a formula that connects different bits of global and local program variables in an inscrutable way. No human developer would fix a bug in such a way. Even if such a cryptic repair renders the program correct, it is of little use. Since the specification may be incomplete, the user cannot judge if it also satisfies unspecified aspects of the design intent. Moreover, such a repair is not maintainable: the user cannot extend the repaired program, refactor it, or fix other bugs without running the risk of breaking the synthesized repair.

Keeping the user in the loop. We envision a program repair tool that does not override the user but keeps the user in the loop. The tool suggests repairs, but it is always the user who selects one. The goal is to reduce the manual effort for finding a correction, not to relieve the user from being responsible for the fix. Furthermore, if only unsatisfactory repairs are presented, the user should be able to refine the specification in order to rule them out. Readability and fine-grainedness of repairs are a crucial usability feature to enable this.

False-positives and false-negatives. Since the user is in the loop anyway, we do not strictly require our debugging approach to be sound or complete. That is, our approach may produce diagnoses and repairs that are incorrect with respect to the specification (we call this a *false-positive*), and it may miss diagnoses and repairs (called a *false-negative*). Recall from Section 2.8.1 that the problem of deciding whether a program satisfies its specification is undecidable in general. Thus, a fully automatic repair algorithm that has no false-positives and no false-negatives cannot exist. While our goal is to keep the amount of false-positives and false-negatives low, our approach will provide many parameters to trade accuracy for efficiency.

4.2 Program Analysis

This section discusses our approach for program analysis, which has the purpose of collecting diagnostic information about the program correctness in the form of formulas. As illustrated in Figure 4.1, this diagnostic information $\mathcal{D}$ will then be used by our automatic fault localization and correction methods. Our program analysis approach consists of two steps. The first step is to preprocesses the program P. Based on a fault model, the preprocessing step identifies program parts that could be faulty, and marks them in the source code. In a second step, symbolic or concolic execution is applied to the preprocessed program $\hat{P}$ in order to obtain the diagnostic information $\mathcal{D}$. Since software program verification is undecidable in general (see Section 2.8.1), the program analysis will be performed in an incomplete and approximative way, producing only approximate correctness information in general.

The following three subsections will discuss our fault model, the preprocessing step, and the program analysis using symbolic or concolic execution, respectively.

4.2.1 Fault Model

Our automatic debugging method needs to identify parts of the program P as potentially faulty. Furthermore, it shall suggest replacements for these faulty parts. Following the terminology for Model-Based Diagnosis (MBD), introduced in Section 2.9, we will refer to individual program parts as *components* of the program. We decompose the program into individual components using a fault model, which describes assumptions about the faults in the program.

Objectives. A good fault model needs to balance conflicting objectives: it should be general in the sense that it covers many faults. It should be fine-grained in order to enable pinpointing potential fault locations precisely. (Since we will later synthesize new implementations for faulty components, a fine-grained component definition also reduces the amount of changed code in the synthesized repairs.) Furthermore, the fault model should allow for efficient debugging algorithms. Clearly, these properties cannot all be maximized at the same time.

Our choice: faulty expressions. As a trade-off between these objectives, we assume that expressions in the program are the only components that can be erroneous. The rest of the program is deemed correct and thus unmodifiable by our debugging approach. This means that our approach cannot handle certain kinds of faults, e.g., wrong control flow structures, missing statements, or faults in the left-hand side of an assignment. Yet, even for such kinds of faults, our approach may find repairs that work around the limitations of the model and fix the program by modifying some expressions in the code. The fault model of incorrect expressions is fine-grained, relatively generic, and has already been used successfully in other debugging approaches [106, 131]. Furthermore, it enables efficient algorithms because the unknown result of an erroneous expression can be handled symbolically, just like an input.

Extensions. In principle, our fault model can be extended in various ways. However, these extensions generally come at the cost of increased computational efforts in program analysis, fault localization and correction. For instance, missing assignment statements can be handled by adding vacuous assignments of the form `v := v;` between all statements and for all variables `v` that are in scope at the respective position. Our debugging approach can then synthesize new expressions for the right-hand side (RHS) of certain assignments, which amounts to adding the so constructed non-vacuous assignment statements to the program. Faults on the left-hand side of an assignment (where the correct expression is assigned to a wrong variable) can be handled by introducing a `switch`-statement with cases that assign the expression to all different variables that are in scope. The choice is set to the case that is implemented in the program, but this choice can be repaired by the debugging tool. Incorrect control flow structures (e.g., the programmer wrote an `if` instead of a `while`) can be handled similarly, in principle.

Alternatives. Alternative fault models for fault localization and correction include mutation-based fault models [74, 179]. A mutation is a small modification of a program's source code. Mutation-based fault models were originally introduced in mutation testing to assess the quality of a test suite [123]: A test suite is considered to be of high quality if it detects many of these small modifications. In mutation-based debugging [74, 179], this idea is turned on its head: an incorrect program is mutated in the hope that the mutation renders the program correct. Mutation-based fault models are usually restricted to small syntactic changes. In contrast, our fault model of incorrect expressions can also handle bugs that can only be fixed with more substantial changes in the code. Another approach is to use fault patterns, which are sequences of statements or constructions that are known to cause defects. They can be defined, e.g., in the form of regular expressions and also allow for efficient fault localization. Pattern-based fault models are popular in static analysis tools for security properties such as Microsoft ESP [71] with its pattern language OPAL [151]. Their disadvantages are that they operate mostly on the syntactic level, that defining fault patters is a lot of (manual) work, and that they can never be exhaustive.

4.2.2 Preprocessing

Based on our fault model of incorrect expressions, we now construct a preprocessed version $\hat{P}$ of the original program P by performing simple textual substitutions. The purpose of the preprocessing step is to express that expressions in the program code may be erroneous. We distinguish two kinds of expressions: *arithmetic expressions* evaluate to some integer value, and *conditional expressions* evaluate to either true or false. Every arithmetic expression `expr` in the code is textually replaced by

```
cmpa(c, expr, v1, v2, ..., vn).
```

Here, `cmpa` is a special function that returns an integer and marks an arithmetic component of the program, `c` is (a unique identifier of) the component $c \in \mathsf{CMP}$, and `v1`, ..., `vn` are the variables that are in scope when component c is executed. Analogously, every conditional expression `expr` in the program P is textually replaced by `cmpc(c, expr, v1, v2, ..., vn)`. The only difference is that `cmpc` returns a Boolean value instead of an integer.

Intuition. Intuitively, we can think of `cmpa` or `cmpc` as a shorthand (in C syntax) for

```
assumeCorrect(c) ? expr : rep_c(v1, ...,vn);.
```

If component c is assumed to be correct, there is no need to modify it. Otherwise it can be replaced by a new expression, which is represented by the (yet unknown) function `rep_c`. The fault localization step will find out which components are assumed to be correct or incorrect. The repair step will compute implementations of the functions `rep_c` for all incorrect components. Note, however, that this is just a conceptual model and we do not introduce such conditional operators into the program explicitly. One reason is that we will later user symbolic or concolic execution on the preprocessed program, where additional conditionals in the code would amplify the path explosion problem (see Section 2.8.2).

Example 22. Consider the following program in C syntax, which will also serve as a running example.

```
1 int max(int a, int b) {
2   int r = a;
3   if(b > a)
4     r = a + 2;
5   assert(r >= a && r >= b);
6   return r;
7 }
```

```
1 int max(int a, int b) {
2   int r = cmpa(1, a, a, b);
3   if(cmpc(2, b>a, a, b, r))
4     r = cmpa(3, a+2, a, b, r);
5   assert(r >= a && r >= b);
6   return r;
7 }
```

The left-hand side shows a program P that is supposed to compute the maximum of two integer numbers `a` and `b`, but contains a bug in Line 4, which should read "`r = b;`". The specification φ is given with the `assert`-statement in Line 5. Note that the specification is incomplete: it only requires that the result cannot be smaller than `a` or `b`, but not that the result must be equal to either `a` or `b`. Still, φ is violated by P: e.g., for `a` $= 0$ and `b` $= 3$, we have that `r` $= 2$, which violates `r >= b`. With our fault model of incorrect expressions, three potentially erroneous components $\mathsf{CMP} = \{c_1, c_2, c_3\}$ are identified: c_1 is the arithmetic expression "`a`" in Line 2, c_2 is the conditional expression "`b > a`" in Line 3, and c_3 is the arithmetic expression "`a + 2`" in Line 4. The right-hand side shows the preprocessed version $\hat{P}$. All expressions have been substituted by calls to `cmpa` or `cmpc` as explained above. The intuitive meaning of Line 4, for instance, is that `r` is assigned `a+2` if component c_3 is assumed to be correct. Otherwise, `r` is set to `rep_`c_3`(a, b, r)`, where `rep_`c_3 is a yet unknown function that can be defined by our debugging tool as a replacement for c_3.

4.2.3 Symbolic Program Analysis

As a final step in our program analysis approach, we now analyze the preprocessed program $\hat{P}$ using a customized version of symbolic or concolic execution to obtain diagnostic information $\mathcal{D}$ for fault localization and correction. Recall from Section 2.8.2 that standard symbolic execution interprets the program using symbols for the inputs. Just like variables, symbols are placeholders that can take any value from a given domain. In our setting, we will administer two sets of symbols: input symbols to represent the unknown input values and repair symbols to represent the results of component executions. This allows us to reason about program correctness even under the assumption that component implementations can be changed in a yet unknown way.

In the following, we first discuss the outcome of the program analysis, namely the diagnostic information $\mathcal{D}$. Section 4.2.3.2 will then explain how this diagnostic information can be computed using symbolic execution. Section 4.2.3.3 will present an alternative realization using concolic execution.

4.2.3.1 Diagnostic Information

Some notation. We assume that all symbols are taken from a sufficiently large set S. Since all our program variables are of the same type (an assumption made to simplify the presentation; see Section 4.1.1), symbols will range either over some domain $\mathbb{D}$ of values or over the Boolean domain $\mathbb{B}$. Depending on the theory used for SMT solving, $\mathbb{D}$ may be the set $\mathbb{B}^n$ of bitvectors of some fixed length n or the integer domain $\mathbb{Z}$. An extension to different domains for different symbols in order to support multiple data types is conceptually simple. We write $\mathbb{F}$ to denote the set of quantifier-free formulas that can be constructed over symbols from S. We will use formulas over the symbols to represent conditions in the program. Similarly, the set of terms over S will be denoted by $\mathbb{T}$. Terms will be used to represent values of program variables as a function over the (unknown) values of the input symbols and the repair symbols. Given that $t_1, t_2 \in \mathbb{T}$ and $F_1, F_2 \in \mathbb{F}$, we assume that $t_1 = t_2$, $t_1 \leq t_2$, $t_1 \geq t_2$, $F_1 \vee F_2$, $F_1 \wedge F_2$, and $\neg F_1$ are in $\mathbb{F}$ as well, with the expected semantics. Finally, we assume that an SMT solver can decide the satisfiability of formulas from $\mathbb{F}$ modulo some chosen background theory.

Definition 23 (Diagnostic Information)**.** *The* diagnostic information $\mathcal{D} = \big(\mathsf{CMP}, \mathrm{TypOf}, \mathrm{Vars}, \bar{\imath}, \bar{r}_{\mathbb{D}}, \bar{r}_{\mathbb{B}}, \mathrm{CmpOf}, \mathrm{Org}_{\mathbb{D}}, \mathrm{Org}_{\mathbb{B}}, \mathrm{Vals}, \mathsf{Correct}(\bar{\imath}, \bar{r})\big)$ *is a tuple where the elements have the following meaning.*

- CMP *is the set of components in the program. Recall that we consider all expressions in the code as components.*
- $\mathrm{TypOf} : \mathsf{CMP} \to \{\mathbb{D}, \mathbb{B}\}$ *is a function that maps a component to its type. An arithmetic component c (marked by a call to* `cmpa` *in $\hat{P}$) has* $\mathrm{TypOf}(c) = \mathbb{D}$*, a conditional component c (calling* `cmpc` *in $\hat{P}$) has* $\mathrm{TypOf}(c) = \mathbb{B}$.
- $\mathrm{Vars} : \mathsf{CMP} \to \mathbb{S}^*$ *is a function that maps each component to a list of strings, which represent the names of the variables that are in scope when the respective component is executed.*
- $\bar{\imath} = (i_1, \ldots, i_k) \in S^k$ *is a vector of input symbols, all ranging over the domain $\mathbb{D}$.*
- $\bar{r}_{\mathbb{D}} = (r_1, \ldots, r_l) \in S^l$ *is a vector of repair symbols, each one ranging over $\mathbb{D}$ and representing the result of an execution of some arithmetic component (marked by a call to* `cmpa` *in $\hat{P}$).*
- $\bar{r}_{\mathbb{B}} = (r_{l+1}, \ldots, r_{l+m}) \in S^m$ *is another vector of repair symbols, all ranging over $\mathbb{B}$ and representing the result of an execution of some conditional component (a call to* `cmpc` *in $\hat{P}$).*
- $\bar{r} = \bar{r}_{\mathbb{D}} \cup \bar{r}_{\mathbb{B}}$ *is an abbreviation for the vector of all repair symbols.*
- $\mathrm{CmpOf} : \{r_1, \ldots, r_{l+m}\} \to \mathsf{CMP}$ *is a function that maps each repair symbol $r_j \in \bar{r}$ to the component that produced it.*
- $\mathrm{Org}_{\mathbb{D}} : \{r_1, \ldots, r_l\} \to \mathbb{T}$ *maps all arithmetic repair symbols $r_j \in \bar{r}_{\mathbb{D}}$ to the symbolic value $t \in \mathbb{T}$ that would be produced by the unmodified component* $\mathrm{CmpOf}(r_j)$.
- *Analogously,* $\mathrm{Org}_{\mathbb{B}} : \{r_{l+1}, \ldots, r_{l+m}\} \to \mathbb{F}$ *maps all conditional repair symbols $r_j \in \bar{r}_{\mathbb{B}}$ to the symbolic truth value $F \in \mathbb{F}$ that is produced by the unmodified component* $\mathrm{CmpOf}(r_j)$.

- $\mathrm{Vals} : \{r_1, \ldots, r_{l+m}\} \to \mathbb{T}^*$ *maps all repair symbols* $r_j \in \overline{r}$ *to a vector of terms that represent the symbolic values of all program variables in scope when* $\mathrm{CmpOf}(r_j)$ *was executed to produce* r_j.
- $\mathsf{Correct}(\overline{i}, \overline{r}) \in \mathbb{F}$ *is a formula over the input symbols* $\overline{i}$ *and the repair symbols* $\overline{r}$ *expressing under which circumstances the preprocessed program* $\hat{P}$ *satisfies its specification* φ.

Intuition. The intuition behind this definition is as follows. Whenever, a component $c \in \mathsf{CMP}$ (i.e., an expression in the program) is executed, a new repair symbol r is created to represent the resulting value. One component can be executed multiple times (e.g., if contained in a loop or in a function that is called multiple times). Hence, the there is a one-to-many relationship between components and repair symbols. This relationship is stored in CmpOf. On the other hand, all repair symbols produced by the same component c have the same type $\mathrm{TypOf}(c)$. A repair symbol r can take any value from the respective domain $x = \mathrm{TypOf}(\mathrm{CmpOf}(r))$. The value that would be produced by the original implementation of the component is stored separately in $\mathrm{Org}_x(r)$. This way, later steps in our debugging flow can extend the correctness formula $\mathsf{Correct}(\overline{i}, \overline{r})$ with constraints of the form $r = \mathrm{Org}_x(r)$ to reason about the program correctness while assuming that (certain) components are left untouched. Furthermore, the repair engine can find different assignments to the repair symbols when searching for a repair. These different assignments may not only be constants, but can be computed from the values $\mathrm{Vals}(r)$ of the program variables at the respective point in the execution. Vars stores the names of the program variables such that computed repairs can later be printed for the user.

Computation of $\mathcal{D}$. The set CMP and the functions TypOf and Vars can easily be computed during the preprocessing step presented in the previous subsection. The computation of the other elements will be discussed in the following. We will first describe a method based on symbolic execution. Section 4.2.3.3 will then present a variant using concolic execution.

4.2.3.2 Symbolic Execution

In order to compute diagnostic information $\mathcal{D}$ according to Definition 23, we perform a symbolic execution of the preprocessed program $\hat{P}$ with the following peculiarities.

Inputs. Whenever a call to the function `input` is encountered, and for all parameters of the entry function of the program, a fresh symbol i is taken from S and appended to $\overline{i}$. The symbol i, interpreted as a term, represents the unknown value of the input.

Components. Whenever a call to the function `cmpa(c, expr, v1, v2, ..., vn)` is executed, a fresh symbol r is taken from S and appended to $\overline{r}_{\mathbb{D}}$. The symbolic value returned by `cmpa` is the term r. In addition, we set $\mathrm{CmpOf}(r)$ to c, $\mathrm{Org}_{\mathbb{D}}(r)$ to the symbolic value of the program expression that is passed as argument `expr`, and $\mathrm{Vals}(r)$ to the tuple of symbolic values of the variables `v1`, ..., `vn`. Calls to `cmpc` are handled similarly, but the new symbol r is appended to $\overline{r}_{\mathbb{B}}$ instead of $\overline{r}_{\mathbb{D}}$, the returned value is the formula r, and $\mathrm{Org}_{\mathbb{B}}(r)$ is set to a formula rather than a term.

Assertions. An assertion `assert(c)` is handled as if it was a shortcut for `if(!c) exitErr();`. That is, the symbolic execution forks into two branches, one where `c` is true and one where `c` is false. In the true-branch, the execution continues normally with the next statement. In the false-branch, it ends and is marked with a specification violation.

Assumptions. An assumption `assume(c)` is handled as a shorthand for `if(!c) exitOK();`. Thus, the symbolic execution forks into two branches again. On the branch where `c` is true, the execution simply continues with the next statement. On the branch where `c` is false, the symbolic execution terminates as if the end of the program would have been reached.

Computation of $\mathsf{Correct}(\overline{i}, \overline{r})$. With the peculiarities explained in the previous paragraphs, we execute the preprocessed program with a user-defined limit on the number and the length of execution paths to consider. Once the symbolic execution is finished, the leaf nodes of the symbolic execution tree are divided into three sets: FAIL is the set of all nodes that are marked with a specification violation, i.e., that ended in `exitErr()` due to an assertion violation. PASS is the set of all nodes where the program terminated normally by reaching the end of the entry function or in `exitOK()` due to an assumption violation. Finally, OPEN contains all the remaining leaf nodes where the execution was aborted because some analysis depth limit was exceeded. Let $\mathsf{PC}_n(\bar{i},\bar{r}) \in \mathbb{F}$ be the path condition of node n in the symbolic execution tree. We define $\mathsf{Correct}(\bar{i},\bar{r}) \in \mathbb{F}$ as

$$\mathsf{Correct}(\bar{i},\bar{r}) = \bigwedge_{n\in\mathsf{FAIL}} \neg\mathsf{PC}_n(\bar{i},\bar{r}). \tag{4.1}$$

Recall from Section 2.8.2 that the path condition $\mathsf{PC}_n(\bar{i},\bar{r})$ expresses under which circumstances the node n in the symbolic is reached. $\mathsf{Correct}(\bar{i},\bar{r})$ thus accumulates all circumstances under which symbolic execution did not encounter a specification violation. Also note that the repair symbols $\bar{r}$ are unconstrained in $\mathsf{Correct}(\bar{i},\bar{r})$. The fault localization and correction algorithms will later combine $\mathsf{Correct}(\bar{i},\bar{r})$ with additional constraints that define the repair symbols appropriately.

Definition 24. *The diagnostic information $\mathcal{D}$ is called* precise *if symbolic execution has been applied exhaustively (i.e., $\mathsf{OPEN} = \emptyset$) and without abstracting the program semantics.*

Example 25. For $\hat{P}$ from Example 22 we obtain $\mathsf{CMP} = \{c_1, c_2, c_3\}$,

$$\begin{aligned}
\mathrm{TypOf}(c_1) &= \mathbb{D}, & \mathrm{TypOf}(c_2) &= \mathbb{B}, & \mathrm{TypOf}(c_3) &= \mathbb{D},\\
\mathrm{Vars}(c_1) &= (\mathtt{a},\mathtt{b}), & \mathrm{Vars}(c_2) &= (\mathtt{a},\mathtt{b},\mathtt{r}), & \mathrm{Vars}(c_3) &= (\mathtt{a},\mathtt{b},\mathtt{r}),\\
\bar{i} &= (\alpha,\beta), & \bar{r}_{\mathbb{D}} &= (\rho_1,\rho_3), & \bar{r}_{\mathbb{B}} &= (\rho_2),\\
\mathrm{CmpOf}(\rho_1) &= c_1, & \mathrm{CmpOf}(\rho_2) &= c_2, & \mathrm{CmpOf}(\rho_3) &= c_3,\\
\mathrm{Org}_{\mathbb{D}}(\rho_1) &= \alpha, & \mathrm{Org}_{\mathbb{B}}(\rho_2) &= \beta > \alpha, & \mathrm{Org}_{\mathbb{D}}(\rho_3) &= \alpha + 2,\\
\mathrm{Vals}(\rho_1) &= (\alpha,\beta), & \mathrm{Vals}(\rho_2) &= (\alpha,\beta,\rho_1), & \mathrm{Vals}(\rho_3) &= (\alpha,\beta,\rho_1), \text{ and}
\end{aligned}$$

$$\mathsf{Correct}(\bar{i},\bar{r}) = (\rho_2 \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee \big((\neg\rho_2) \wedge \rho_1 \geq \alpha \wedge \rho_1 \geq \beta\big)$$

Lemma 26. *Let $\mathbf{i} \in \mathbb{D}^k$, $\mathbf{r}_{\mathbb{D}} \in \mathbb{D}^l$ and $\mathbf{r}_{\mathbb{B}} \in \mathbb{B}^m$ be three vectors of concrete symbol values. If $\mathcal{D}$ is precise (Definition 24), then the condition $\mathsf{Correct}(\mathbf{i}, \mathbf{r}_{\mathbb{D}} \cup \mathbf{r}_{\mathbb{B}})$ is* true *iff $\hat{P}$ fulfills its specification, given that $\mathbf{i}$ is used as input vector and $\mathbf{r}_{\mathbb{D}}$ and $\mathbf{r}_{\mathbb{B}}$ is the vector of values returned by calls to the function* `cmpa` *and* `cmpc`*, respectively.*

Proof. Every leaf node n in the symbolic execution tree represents an execution path through the program (the sequence of statements encountered when traversing from the root node to n). Let n_e be the node that represents the execution path p_e that is induced by the values $\mathbf{i}, \mathbf{r}_{\mathbb{D}}, \mathbf{r}_{\mathbb{B}}$. Since all leaf nodes have disjunct path conditions (i.e., $\mathsf{PC}_{n_1} \wedge \mathsf{PC}_{n_2}$ is unsatisfiable for all pairs of distinct leaf nodes $n_1 \neq n_2$), $\mathsf{PC}_n(\mathbf{i}, \mathbf{r}_{\mathbb{D}} \cup \mathbf{r}_{\mathbb{B}})$ is true iff $n = n_e$. Lemma 26 holds because $\neg\mathsf{PC}_{n_e}(\mathbf{i}, \mathbf{r}_{\mathbb{D}} \cup \mathbf{r}_{\mathbb{B}}) = \mathsf{false}$ is a conjunct of $\mathsf{Correct}(\mathbf{i}, \mathbf{r}_{\mathbb{D}} \cup \mathbf{r}_{\mathbb{B}})$ iff $n_e \in \mathsf{FAIL}$, i.e., iff p_e violates the specification. □

Proposition 27. *Let P be a potentially incorrect program and let $\mathbf{i} \in \mathbb{D}^k$ be an input vector. Assuming that $\mathcal{D}$ is precise (Definition 24), the formula*

$$\exists \bar{r} : \mathsf{Correct}(\mathbf{i},\bar{r}) \wedge \bigwedge_{r\in\bar{r}_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r)(\mathbf{i},\bar{r}) \wedge \bigwedge_{r\in\bar{r}_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r)(\mathbf{i},\bar{r}) \tag{4.2}$$

is true *iff P fulfills the specification φ when executed with input* $\mathbf{i}$.

Proof. Let $\mathbf{r}_{\mathbb{D}} \in \mathbb{D}^l$ be concrete values returned by calls to `cmpa` in $\hat{P}$, let $\mathbf{r}_{\mathbb{B}} \in \mathbb{B}^m$ be the concrete values returned by `cmpc`, and let $\mathbf{r} = \mathbf{r}_{\mathbb{D}} \cup \mathbf{r}_{\mathbb{B}}$. According to Lemma 26, $\mathsf{Correct}(\mathbf{i}, \mathbf{r})$ is true iff $\hat{P}$ fulfills the specification φ when executed with $\mathbf{i}, \mathbf{r}_{\mathbb{D}}, \mathbf{r}_{\mathbb{B}}$. The conjuncts in Formula 4.2 require that all components in $\hat{P}$ return the same value as the respective expressions in P. Hence, Formula 4.2 is true iff P fulfills φ for input $\mathbf{i}$. □

Lemma 26 states how the formula $\mathsf{Correct}(\bar{i},\bar{r})$ can be used to make statements about the correctness of the preprocessed program $\hat{P}$, depending on the inputs and the components. Proposition 27 establishes the link to the correctness of the original program P using the information stored in $\mathrm{Org}_{\mathbb{D}}$ and $\mathrm{Org}_{\mathbb{B}}$.

4.2.3.3 Concolic Execution

This section explains how concolic execution can be used for program analysis as an alternative to using symbolic execution. Recall from Section 2.8.3 that concolic execution is just a variant of symbolic execution where the program is executed using concrete values and symbols at the same time. The analysis proceeds in several runs. In each run, the program is executed using concrete inputs, while the symbolic execution for the activated execution path is performed in parallel.

Computing the diagnostic information $\mathcal{D}$. Using concolic execution instead of symbolic execution to compute $\mathcal{D} = \big(\mathsf{CMP}, \mathrm{TypOf}, \mathrm{Vars}, \bar{\imath}, \bar{r}_{\mathbb{D}}, \bar{r}_{\mathbb{B}}, \mathrm{CmpOf}, \mathrm{Org}_{\mathbb{D}}, \mathrm{Org}_{\mathbb{B}}, \mathrm{Vals}, \mathsf{Correct}(\bar{\imath}, \bar{r})\big)$ is simple. $\mathcal{D}$ is stored persistently across all execution runs. The elements $\bar{\imath}$, $\bar{r}_{\mathbb{D}}$, $\bar{r}_{\mathbb{B}}$, CmpOf, $\mathrm{Org}_{\mathbb{D}}$, $\mathrm{Org}_{\mathbb{B}}$ and Vals are initially empty and updated as for symbolic execution while calls to the functions `input`, `cmpa` and `cmpc` are carried out. At the end of each concolic execution run, the negation $\neg\mathsf{PC}(\bar{\imath}, \bar{r})$ of the resulting path condition is conjoined to the correctness formula $\mathsf{Correct}(\bar{\imath}, \bar{r})$ if the program terminated in `exitErr()` due to an assertion violation. Otherwise, Correct is not updated. This procedure computes $\mathsf{Correct}(\bar{\imath}, \bar{r})$ as in Equation 4.1, but the computation is done on the fly rather than a posteriori.

Managing symbols. A simple realization would create symbols that are fresh across all execution runs, resulting in disjoint symbol sets for different runs. However, this has the disadvantage that different symbols may be used to represent the same unknown value. In particular, many of the execution runs may share the same prefix of executed statements, on which the same symbols can be reused in order to simplify $\mathcal{D}$. In our realization, we reuse input symbols from previous execution runs based on the order in which they are read by the program: the j^{th} input that is consumed by the program is always represented by the same input symbol $i_j \in \bar{\imath}$ in all runs. This way, we can think of the input symbols as being read from a tape while the program is being executed. When executing calls to `cmpa`, we reuse a symbol $r_x \in \bar{r}_{\mathbb{D}}$ instead of producing a new symbol r_y if $\mathrm{CmpOf}(r_x) = \mathrm{CmpOf}(r_y)$, $\mathrm{Org}_{\mathbb{D}}(r_x) = \mathrm{Org}_{\mathbb{D}}(r_y)$, and $\mathrm{Vals}(r_x) = \mathrm{Vals}(r_y)$.[2] Existing symbols from $\bar{r}_{\mathbb{B}}$ are reused in calls to `cmpc` analogously.

Advantages. As described above, concolic execution can compute the same diagnostic information as symbolic execution. In particular, Lemma 26 and Proposition 27 hold analogously. Advantages of concolic execution over symbolic execution (less memory consumption, flexibility in abstracting program semantics by using concrete variable values) have already been discussed in Section 2.8.3.

4.3 SMT-Based Fault Localization

Based on the diagnostic information $\mathcal{D}$ computed with the program analysis approach from the previous section, this section now discusses how to compute diagnoses, which are sets of program components that could be responsible for the incorrectness. As illustrated in Figure 4.1, these diagnoses are on the one hand presented to the user as potential fault locations, and on the other hand used as input for our fault correction method to synthesize repairs.

Our fault localization method rests upon Model-Based Diagnosis (MBD), as introduced in Section 2.9. Standard MBD takes as input a model of a system together with a contradicting observation. The contradiction manifests in conflicts, which need to be explained using diagnoses. In our setting, a program conflicts with its specification, so we need a different notion of a conflict. Deriving diagnoses from conflicts then works in the standard way. In the following, we will first define a notion of conflicts and diagnoses in our setting. Section 4.3.2 will then discuss how they can be computed efficiently.

4.3.1 A Definition for Conflicts and Diagnoses

We define a function $\mathsf{repairable} : 2^{\mathsf{CMP}} \rightarrow \mathbb{B}$. Intuitively, $\mathsf{repairable}(A)$ maps a set $A \subseteq \mathsf{CMP}$ of components to true iff program $\hat{P}$ can be repaired for all inputs, assuming that all components $c \in A$ are

[2] The comparison is performed syntactically in order to reduce the computational overhead. That is, if two terms in $\mathrm{Org}_{\mathbb{D}}$ or Vals are syntactically different but semantically equivalent (e.g., $a+b$ vs. $b+a$), a new symbol may be produced unnecessarily.

correct and need not be modified. Formally, we define

$$\mathsf{repairable}(A) = \forall \bar{i} : \exists \bar{r} : \mathsf{Correct}(\bar{i}, \bar{r}) \wedge \bigwedge_{r \in R_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r) \wedge \bigwedge_{r \in R_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r), \tag{4.3}$$

where $R_{\mathbb{D}}$ stands for $\{r \in \bar{r}_{\mathbb{D}} \mid \mathrm{CmpOf}(r) \in A\}$ and $R_{\mathbb{B}} = \{r \in \bar{r}_{\mathbb{B}} \mid \mathrm{CmpOf}(r) \in A\}$. The definition says that a program is repairable iff, for all inputs, there exist values that can be returned by the components (the functions `cmpa` and `cmpc` in $\hat{P}$) such that Correct is true, which means that the specification is fulfilled. Components that are assumed to be correct can only return the value that would be returned by the original version of that component. This is enforced by the conjuncts of the form $r = \mathrm{Org}_{\mathbb{D}}(r)$ and $r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r)$ if $\mathrm{CmpOf}(r) \in A$. The other components can return arbitrary values.

Lemma 28. *The function* repairable *is monotonic in the sense that, for all component sets* $A' \subseteq A \subseteq$ CMP, *we have that* repairable(A) *implies* repairable(A').

Proof. Removing elements from A only removes conjuncts Equation 4.3. □

Definition 29. *A set* $\Delta \subseteq$ CMP *is a* diagnosis *for program* P *iff* repairable(CMP $\setminus \Delta$) $=$ true. *A set* $C \subseteq$ CMP *is a* conflict *iff* repairable$(C) =$ false.

A diagnosis is a set Δ of components that can be modified such that P becomes correct. The reason is that, for every input, it is possible to find some value that can be returned by the components $c \in \Delta$ such that the specification is fulfilled. Hence, diagnoses represent fault candidates. A conflict is a set C of components from which at least one component has to be modified in order to obtain a correct program. If none of the components $c \in C$ are modified, the program will be incorrect not matter what happens with components $c \notin C$.

Example 30. Consider the program from Example 22 with the diagnostic information $\mathcal{D}$ from Example 25. In summary, we have the following repairability situation:

Case	Set $A \subseteq$ CMP	repairable(A)	Diagnosis	Conflict
1	$\emptyset$	true	$\{c_1, c_2, c_3\}$	
2	$\{c_1\}$	true	$\{c_2, c_3\}$	
3	$\{c_2\}$	true	$\{c_1, c_3\}$	
4	$\{c_3\}$	true	$\{c_1, c_2\}$	
5	$\{c_1, c_2\}$	true	$\{c_3\}$	
6	$\{c_1, c_3\}$	false		$\{c_1, c_3\}$
7	$\{c_2, c_3\}$	false		$\{c_2, c_3\}$
8	$\{c_1, c_2, c_3\}$	false		$\{c_1, c_2, c_3\}$

That is, the minimal diagnoses are $\Delta_1 = \{c_3\}$ and $\Delta_2 = \{c_1, c_2\}$. Intuitively, this means that component c_3 can be modified such that P becomes correct. Alternatively, both c_1 and c_2 can be modified simultaneously to fix the problem. Since repairable is monotonic (Lemma 28), all supersets of Δ_1 and Δ_2 are diagnoses as well. The minimal conflicts are $C_1 = \{c_1, c_3\}$ and $C_2 = \{c_2, c_3\}$, and all supersets are conflicts as well. We will now discuss a few of the cases in more detail. Case 5 establishes that the component set $\Delta_1 = \{c_3\}$ is a diagnosis because

$$\begin{aligned}
\mathsf{repairable}(\{c_1, c_2\}) &= \forall \alpha, \beta : \exists \rho_1, \rho_2, \rho_3 : && \big((\rho_2 \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee ((\neg\rho_2) \wedge \rho_1 \geq \alpha \wedge \rho_1 \geq \beta)\big) \wedge \\
& && (\rho_1 = \alpha) \wedge \big(\rho_2 \leftrightarrow (\beta > \alpha)\big) \\
&= \forall \alpha, \beta : \exists \rho_3 : && (\beta > \alpha \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee (\beta \leq \alpha \wedge \alpha \geq \alpha \wedge \alpha \geq \beta) \\
&= \forall \alpha, \beta : \exists \rho_3 : && (\beta > \alpha \wedge \rho_3 \geq \beta) \vee (\beta \leq \alpha) \\
&= \mathsf{true} && \text{(because we can set } \rho_3 = \beta\text{).}
\end{aligned}$$

The intuitive reason is that we can always find some alternative value ρ_3 to return when executing component c_3 (namely the value β of the second input b) in order to make Correct evaluate to true, which means to satisfy the specification. Hence, c_3 may be responsible for the incorrectness of P — it is a diagnosis. On the other side, Case 6 states that $C_1 = \{c_1, c_3\}$ is a conflict because

$$\begin{aligned}
\mathsf{repairable}(\{c_1, c_3\}) &= \forall \alpha, \beta : \exists \rho_1, \rho_2, \rho_3 : \big((\rho_2 \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee ((\neg\rho_2) \wedge \rho_1 \geq \alpha \wedge \rho_1 \geq \beta)\big) \wedge \\
& (\rho_1 = \alpha) \wedge (\rho_3 = \alpha + 2) \\
&= \forall \alpha, \beta : \exists \rho_2 : \quad (\rho_2 \wedge \alpha + 2 \geq \alpha \wedge \alpha + 2 \geq \beta) \vee \big((\neg\rho_2) \wedge \alpha \geq \alpha \wedge \alpha \geq \beta\big) \\
&= \forall \alpha, \beta : \exists \rho_2 : \quad (\rho_2 \wedge \alpha + 2 \geq \beta) \vee \big((\neg\rho_2) \wedge \alpha \geq \beta\big) \\
&= \mathsf{false} \quad \text{because with } \alpha = 0 \text{ and } \beta = 3 \text{ no value for } \rho_2 \text{ works.}
\end{aligned}$$

This means that c_1 and c_3 cannot both be correct — at least one of them must be modified in order to obtain a correct program. This implies that the set $\{c_2\} = \mathsf{CMP} \setminus C_1$ cannot be a diagnosis.

4.3.2 Computation of Conflicts and Diagnoses

The following theorem, which is a slight adaptation of Theorem 4.4 from the work of Reiter [182], states that minimal diagnoses can be computed as minimal hitting sets for the collection of conflicts.

Theorem 31. *A set $\Delta \subseteq \mathsf{CMP}$ of components is a minimal diagnosis for program P iff it is a minimal hitting set for the collection $\mathcal{K}$ of conflicts for $\hat{P}$.*

Proof. Using Lemma 28, the proof in the work of Reiter [182] applies. □

Basic algorithm. Based on Theorem 31, we apply the hitting set tree algorithm of Reiter [182] (with the fix by Greiner et al. [103]) to compute diagnoses. This algorithm requires a procedure to compute a conflict without components from a given set $N \subseteq \mathsf{CMP}$, if such a conflict exists. Such a procedure can be implemented as

$$\mathsf{getConfWo}(N) = \begin{cases} \mathsf{CMP} \setminus N & \text{if } \mathsf{repairable}(\mathsf{CMP} \setminus N) = \mathsf{false}\text{, and} \\ \text{“None”} & \text{otherwise.} \end{cases}$$

That is, getConfWo in turn requires a procedure to evaluate repairable, as defined in Equation 4.3.

Avoiding quantifier alternations. Deciding repairability as defined in Equation 4.3 is computationally hard or, depending on $\mathbb{D}$, $\mathbb{F}$ and the background theory, even undecidable. One reason is the quantifier alternation. As an approximation, we therefore propose to check repairability only for a given set $J \subseteq 2^{(\mathbb{D}^k)}$ of input vectors. That is, instead of $\mathsf{repairable}(A)$ we rather consult

$$\mathsf{repairable}'(A) = \bigwedge_{\mathbf{i} \in J} \Big(\exists \overline{r} : \mathsf{Correct}(\mathbf{i}, \overline{r}) \wedge \bigwedge_{r \in R_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r)(\mathbf{i}, \overline{r}) \wedge \bigwedge_{r \in R_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r)(\mathbf{i}, \overline{r})\Big) \quad (4.4)$$

with $R_{\mathbb{D}} = \{r \in \overline{r}_{\mathbb{D}} \mid \mathrm{CmpOf}(r) \in A\}$ and $R_{\mathbb{B}} = \{r \in \overline{r}_{\mathbb{B}} \mid \mathrm{CmpOf}(r) \in A\}$. Note that Equation 4.4 differs from Equation 4.3 only in having a finite conjunction over the inputs instead of a universal quantification. For the set J, we use only inputs $\mathbf{i}$ that make the program P violate its specification φ because for all other inputs, P is trivially repairable. In other words, J consists only of counterexamples to the correctness of P. When applying concolic execution for program analysis, such concrete input values $\mathbf{i}$ are computed anyway. When using symbolic execution, satisfying assignments to path conditions can be computed in order to obtain input vectors for J. The following theorem states that using repairable′ instead of repairable can only lead to false positives but not to missing diagnoses.

Theorem 32. *Every diagnosis Δ with respect to the definition of* repairable *is also a diagnosis with respect to* repairable′.

Algorithm 4.1 GETMINCONFWO: Computes a minimal conflict without elements from N.

```
procedure GETMINCONFWO($N$), returns: A minimal conflict $C \subseteq \mathsf{CMP} \setminus N$ or "None"
    $\mathbf{x} := \mathsf{true}$
    for each component $c \in \mathsf{CMP} \setminus N$ do
        $\mathbf{x} := \mathbf{x} \wedge x_c$
    end for
    if SMTSAT$\big(\mathbf{x} \wedge F(\overline{x}, \dots)\big)$ then
        return "None"
    else
        $\mathbf{x}' :=$ SMTMINUNSATCORE$\big(\mathbf{x}, F(\overline{x}, \dots)\big)$
        return $\{c \mid x_c \in \mathbf{x}'\}$
    end if
end procedure
```

Proof. Since $\mathsf{repairable}(A)$ implies $\mathsf{repairable}'(A)$ for all $A \subseteq \mathsf{CMP}$, we have that $\mathsf{repairable}(\mathsf{CMP} \setminus \Delta)$ implies $\mathsf{repairable}'(\mathsf{CMP} \setminus \Delta)$. □

Answering repairability queries. The quantifier-free part of $\mathsf{repairable}'$ is in $\mathbb{F}$. Therefore, a query $\mathsf{repairable}'(A)$ can be solved using one SMT solver call per input vector $\mathbf{i} \in J$. These individual calls can also be carried out in parallel. An alternative is to swap the conjunction over the input vectors $\mathbf{i} \in J$ with the existential quantification of $\overline{r}$ in Equation 4.4, rename all repair symbols $\overline{r}$ to fresh ones for every conjunct, and use only one SMT solver call. Let $\overline{r}_{\mathbf{i}} = (r_{\mathbf{i}1}, r_{\mathbf{i}2}, \dots)$ be a fresh copy of $\overline{r} = (r_1, r_2, \dots)$ for input vector $\mathbf{i}$. We now have that $\mathsf{repairable}'(A)$ is true iff

$$\bigwedge_{\mathbf{i} \in J} \Big(\mathsf{Correct}(\mathbf{i}, \overline{r}_{\mathbf{i}}) \wedge \bigwedge_{r_j \in R_{\mathbb{D}}} r_{\mathbf{i}j} = \mathrm{Org}_{\mathbb{D}}(r_j)(\mathbf{i}, \overline{r}_{\mathbf{i}}) \wedge \bigwedge_{r_j \in R_{\mathbb{B}}} r_{\mathbf{i}j} \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r_j)(\mathbf{i}, \overline{r}_{\mathbf{i}}) \Big) \qquad (4.5)$$

is satisfiable, where $R_{\mathbb{D}}$ is again defined as $\{r_j \in \overline{r}_{\mathbb{D}} \mid \mathrm{CmpOf}(r_j) \in A\}$ and $R_{\mathbb{B}} = \{r_j \in \overline{r}_{\mathbb{B}} \mid \mathrm{CmpOf}(r_j) \in A\}$.

Computing minimal conflicts. The performance of Reiter's hitting set tree algorithm [182] increases if the computed conflicts are small. A minimal conflict not intersecting with a certain set $N \subseteq \mathsf{CMP}$ of components can be computed in different ways. One way is to use a failure-preserving minimization algorithm like Delta Debugging [213] or QuickXplain [132] to repeatedly invoke $\mathsf{repairable}'$ with different subsets of $\mathsf{CMP} \setminus N$ until a minimal subset for which $\mathsf{repairable}'$ evaluates to false is found. Another option is based on the observation that every minimal conflict corresponds to an unsatisfiable core in Equation 4.5: Every component $c \in A$ activates a certain set B_c of constraints of the form $r = \mathrm{Org}(r)$ in Equation 4.5. We search for a minimal subset $M \subseteq \{B_c \mid c \in A\}$ of these constraints such that the conjunction with $\mathsf{Correct}(\mathbf{i}, \overline{r}_{\mathbf{i}})$ for all $\mathbf{i} \in J$ is still unsatisfiable. To realize this idea, we introduce a vector $\overline{x}$ containing one Boolean activation variable x_c per component $c \in \mathsf{CMP}$. With these activation variables, we construct the formula

$$\begin{aligned} F(\overline{x}, \dots) = \bigwedge_{\mathbf{i} \in J} \mathsf{Correct}(\mathbf{i}, \overline{r}_{\mathbf{i}}) \wedge & \bigwedge_{r_j \in \overline{r}_{\mathbb{D}}} x_{\mathrm{CmpOf}(r_j)} \rightarrow \big(r_{\mathbf{i}j} = \mathrm{Org}_{\mathbb{D}}(r_j)(\mathbf{i}, \overline{r}_{\mathbf{i}})\big) \wedge \\ & \bigwedge_{r_j \in \overline{r}_{\mathbb{B}}} x_{\mathrm{CmpOf}(r_j)} \rightarrow \big(r_{\mathbf{i}j} \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r_j)(\mathbf{i}, \overline{r}_{\mathbf{i}})\big) \end{aligned}$$

That is, setting x_c to true in F intuitively means that component $c \in \mathsf{CMP}$ is correct and cannot be modified. The procedure GETMINCONFWO in Algorithm 4.1 finally uses F to compute minimal conflicts (not intersecting with some set $N \subseteq \mathsf{CMP}$) using unsatisfiable cores computed by an SMT solver. The search for a minimal set $C \subseteq \mathsf{CMP} \setminus N$ of components such that $\mathsf{repairable}'(C) = \mathsf{false}$ is reduced to a search for a minimal subset $\mathbf{x}'$ of the activation literals in $\mathbf{x}$ such that $\mathbf{x}' \wedge F(\overline{x}, \dots)$ is still unsatisfiable.

Note that GETMINCONFWO is also well suited for incremental SMT solving, even across multiple calls to GETMINCONFWO: The formula $F(\overline{x}, \ldots)$ can always remain asserted — only the set of additionally asserted activation literals changes.

Summary. The purpose of the fault localization approach presented in this section is to compute diagnoses based on the diagnostic information $\mathcal{D}$ (see Figure 4.1). A diagnosis $\Delta \subseteq \mathsf{CMP}$ is a set of components of the incorrect program P that can be modified in such a way that P becomes correct. Using the hitting set tree algorithm of Reiter [182], diagnoses can be computed efficiently using a procedure to compute minimal conflicts. Such a procedure is presented in Algorithm 4.1. It approximates the necessary repairability check by replacing a universal quantification with a finite conjunction for the sake of efficiency. This approximation can result in false-positives but not in diagnoses being missed.

4.4 Fault Localization Based on Deductive Verification

The fault localization approach presented in the previous section faces several challenges. First, our specifications are given as assertions in the code and require reasoning about the program correctness on a global level. Second, applying our program analysis technique using symbolic or concolic execution exhaustively may be impracticable, especially in the presence of loops where the number of iterations depends on inputs. Third, computing diagnoses precisely requires solving formulas that contain quantifier alternations. Even if the selected background theory is decidable with quantifiers, this can be challenging for SMT solvers. To counteract that, we turned the universal quantifications into finite conjunctions, but this can result in false-positives.

In this section, we propose an alternative method for fault localization that addresses these issues. We work with specifications that are given in the form of preconditions and postconditions for every function in the program. This allows for local reasoning about program correctness (every function can be analyzed in isolation) and therefore has a higher potential for scaling up to larger programs. Second, we require that loops in the program are annotated with loop invariants. Using these invariants, loops can be analyzed inductively rather than iteratively. Third, we use a first-order theorem prover instead of an SMT solver to reason about repairability without approximating the quantifier alternation.

4.4.1 From Deductive Program Verification to Fault Localization

Our alternative approach for fault localization can be realized with only minor modifications to a typical deductive verification flow.

Deductive verification. Model checking attempts to prove a program correct by exploring all program behaviors exhaustively. In contrast, deductive program verification techniques construct proof obligations, which are formulas that imply program correctness if they can be shown to hold true. One approach for constructing such proof obligations is to use Hoare logic (see Section 2.8.4): Every function of the program is analyzed in isolation. Based on the postcondition Q of a function f and the loop invariants, the Hoare axioms are used to compute the weakest precondition $W(\overline{\imath}) = \mathsf{wp}(f, Q)$ under which the postcondition holds. This weakest precondition $W(\overline{\imath})$ is a formula over the input variables $\overline{\imath}$ of the function under analysis. Given the user-defined precondition $P(\overline{\imath})$ of the function, a correctness formula $\mathsf{Corr}(\overline{\imath}) = P(\overline{\imath}) \rightarrow W(\overline{\imath})$ can then be computed. Finally, an (automatic) theorem prover can be used to check the validity of the proof obligation $\forall \overline{\imath} : \mathsf{Corr}(\overline{\imath})$. This approach (sketched here in simplified form) is realized in the WP plug-in of the widely used software analysis tool suite Frama-C [69].

Fault localization using deductive verification. With our fault model of incorrect expressions, we can use a deductive verification engine (almost) as a black-box in order to perform fault localization. We use the same repairability-based notion of diagnoses and conflicts as in Section 4.3 (Definition 29). In order to decide $\mathsf{repairable}(A)$ for some component set $A \subseteq \mathsf{CMP}$, we

1. textually replace components $c \notin A$ in the program code by fresh repair variables $\bar{r}$,
2. call a deductive verification engine to produce a proof obligation $\forall \bar{\imath}, \bar{r} : \mathsf{Corr}(\bar{\imath}, \bar{r})$, and
3. modify the quantifier structure of the proof obligation in order to check $\forall \bar{\imath} : \exists \bar{r} : \mathsf{Corr}(\bar{\imath}, \bar{r})$ using an automatic theorem prover.

We set $\mathsf{repairable}(A) = \mathsf{true}$ if the theorem prover managed to prove the validity of $\forall \bar{\imath} : \exists \bar{r} : \mathsf{Corr}(\bar{\imath}, \bar{r})$. In all other cases (the theorem prover disproved validity, it had a timeout or returned "Unknown") we set $\mathsf{repairable}(A) = \mathsf{false}$.[3] With this procedure to answer repairability queries, the basic algorithm from Section 4.3.2 can be applied to compute diagnoses.

Example 33. Consider the program from Example 22, but specified with a pre- and a postcondition:

```
  /*@requires true;
    @ensures \result>=a &&
    @         \result>=b;
   */
1 int max(int a, int b) {
2   int r = a;
3   if(b > a)
4     r = a + 2;
6   return r;
7 }
```

```
1 int max(int a, int b) {
    {$(b > a \land \rho_3 \geq a \land \rho_3 \geq b) \lor b \leq a$}
2   int r = a;
    {$(b > a \land \rho_3 \geq a \land \rho_3 \geq b) \lor (b \leq a \land r \geq a \land r \geq b)$}
3   if(b > a) {
      {$\rho_3 \geq a \land \rho_3 \geq b$}
4     r = $\rho_3$;
      {$r \geq a \land r \geq b$}
    } else {
      {$r \geq a \land r \geq b$}
    }
    {$r \geq a \land r \geq b$}
6   return r;
7 }
```

The program is shown on the left. With $\bar{\imath} = (a, b)$, the `@requires` clause defines the precondition $\mathsf{Pre}(\bar{\imath}) = \mathsf{true}$ and the `@ensures` clause sets the postcondition to $\texttt{\result} \geq a \land \texttt{\result} \geq b$, with `\result` being the value returned by `max`. We consider the same components as in Example 22 and apply our approach to check if $\{c_3\}$ is a diagnosis (recall that c_3 is the expression `a+2` in Line 4). Definition 29 states that $\{c_3\}$ is a diagnosis iff $\mathsf{repairable}(\{c_1, c_2\}) = \mathsf{true}$. Our approach for checking $\mathsf{repairable}(\{c_1, c_2\})$ is carried out in the source code listing on the right. The component c_3 is replaced by a fresh repair variable ρ_3 (Line 4). Next, for every point in the program, Hoare logic (see Section 2.8.4) is used to compute the weakest precondition under which the postcondition holds. These preconditions are shown in blue. With $\bar{r} = (\rho_3)$, the weakest precondition for `max` is $\mathsf{Wp}(\bar{\imath}, \bar{r}) = (b > a \land \rho_3 \geq a \land \rho_3 \geq b) \lor b \leq a$. Since $\mathsf{Pre}(\bar{\imath}) = \mathsf{true}$, we have that $\mathsf{Corr}(\bar{\imath}, \bar{r}) = \mathsf{Wp}(\bar{\imath}, \bar{r})$, so

$$\mathsf{repairable}(\{c_1, c_2\}) = \forall a, b : \exists \rho_3 : (b > a \land \rho_3 \geq a \land \rho_3 \geq b) \lor b \leq a.$$

This formula is true because ρ_3 can be set to b. This means that $\{c_3\}$ is a diagnosis.

4.4.2 Discussion

The following paragraphs point out some differences to the fault localization approach from Section 4.3.

Repeated program analysis. Our fault localization approach based on deductive verification builds the correctness formula from the program source code anew for every repairability query. However, this is not considered a severe issue because constructing the correctness formula is merely a syntactic operation and can thus be expected to run fast. The advantage is that deductive program analysis can be applied as a black-box operation. In contrast, the SMT-based fault localization approach presented in Section 4.3 performs program analysis only once, but uses a customized version of symbolic or concolic execution

[3] Categorizing timeouts or "Unknown" verdicts as irreparable cases can result is missed diagnoses, but overall yields more accurate results in our experiments than classifying them as repairable.

to gather global correctness information for different local correctness assumptions about components in one go. Since symbolic (or concolic) execution involves SMT solving, redoing it for every repairability query would be costly.

Preconditions and postconditions versus assertions. In contrast to assertions, using preconditions and postconditions as a specification allows for localized correctness reasoning, i.e., every function in the code can be analyzed in isolation. Besides a scalability advantage, verifying functions in isolation also gives a coarse-grained fault localization as a side product, because the (potentially) erroneous functions are reported. The obvious disadvantage is that annotating every function in the code with contracts and every loop with invariants is a lot of manual work. Since this manual specification work is itself error prone, contracts can easily by faulty or incomplete. However, this issue is orthogonal to this thesis — we always assume specifications to be correct.

First-order theorem provers versus SMT solvers. SMT solvers can be seen as specialized theorem provers, optimized for efficient theory reasoning in quantifier-free formulas. Theorem provers can be a better choice when reasoning about quantified formulas. This motivated our choice of using a theorem prover for discharging the quantified repairability formulas in this section. This being said, we also note that the differences between these two kinds of reasoning engines are fading: SMT solvers are increasingly extended with support for quantifiers, and theorem provers are getting better in theory reasoning.

4.5 Template-Based Repair Synthesis

This section presents our basic approach for synthesizing repairs for an incorrect program P. As illustrated in Figure 4.1, it takes as input the diagnostic information $\mathcal{D}$, computed as explained in Section 4.2, and a diagnosis $\Delta \subseteq \mathsf{CMP}$. The diagnosis is a set of program components that are potentially faulty. Diagnoses can be computed with the fault localization methods from Section 4.3 or 4.4, but may also be given by the user or some other means. If successful, the output of our synthesis procedure is a repair in the form of a modified program P', which differs from P only in the components Δ. Assuming that $\mathcal{D}$ is precise (Definition 24), the repaired program P' will satisfy its specification φ for all inputs. Otherwise, the repaired program will be correct up to the analysis depth and modulo the approximations used during the program analysis phase. If multiple diagnoses are found by the fault localization step, our repair method is executed for each of them.

This section will first discuss how templates can be used to fix the structure of repairs. Subsection 4.5.2 will then present our solution for computing repairs using Counterexample-Guided Inductive Synthesis (CEGIS). Subsection 4.5.3 finally proposes heuristics to speed up the repair computation. An improved variant of our repair synthesis method will be presented in Section 4.7.

4.5.1 Templates

In our software repair approach, we synthesize new expressions for all components $c \in \Delta$ in a given diagnosis Δ. Similar to our template-based method for hardware controller synthesis (Section 3.1.5), we use templates in order to fix the structure of the new expressions to synthesize. This reduces the search for new expressions to the search for constants. In the software setting, templates are now expressions consisting of program variables and template parameters. Concrete values for the template parameters induce a concrete expression over the program variables.

Example 34. The template $k_0 + k_1 \cdot \mathtt{v1} + k_2 \cdot \mathtt{v2}$, where k_0, k_1, k_2 are template parameters and $\mathtt{v1}, \mathtt{v2}$ are program variables, can be instantiated to any linear expression over the variables `v1` and `v2`. The values $k_0 = -2$, $k_1 = 1$, and $k_2 = 0$ induce the expression `v1 - 2`.

Templates also provide control over the expressions subjected to search and thus support our goal of synthesizing readable repairs. In order to obtain simple repairs, we propose to start with simple templates and switch to more expressive templates if no repair is found with the simple ones. In our implementation, we strive for a fully automatic debugging flow and use a fixed set of templates. However, in principle, templates can also be defined by the user.

Creating templates. Recall that we are given a diagnosis $\Delta \subseteq \mathsf{CMP}$ and the diagnostic information $\mathcal{D} = \big(\mathsf{CMP}, \mathrm{TypOf}, \mathrm{Vars}, \overline{i}, \overline{r}_{\mathbb{D}}, \overline{r}_{\mathbb{B}}, \mathrm{CmpOf}, \mathrm{Org}_{\mathbb{D}}, \mathrm{Org}_{\mathbb{B}}, \mathrm{Vals}, \mathsf{Correct}(\overline{i}, \overline{r})\big)$ as defined in Definition 23. For every arithmetic component $c \in \Delta$ with $\mathrm{TypOf}(c) = \mathbb{D}$, we create a template $T_c(\overline{k}_c, \overline{v}_c) \in \mathbb{T}$ as a term over two vectors of fresh symbols $\overline{k}_c \in S^*$ and $\overline{v}_c \in S^{|\mathrm{Vars}(c)|}$. The symbols in $\overline{k}_c$ are template parameters and range over $\mathbb{D}$. The symbols $\overline{v}_c$ represent the values of the program variables in scope when component c is executed. Hence, the symbols $\overline{v}_c$ range over $\mathbb{D}$ as well. Similarly, for all conditional components $c \in \Delta$ with $\mathrm{TypOf}(c) = \mathbb{B}$, we create a template $T_c(\overline{k}_c, \overline{v}_c) \in \mathbb{F}$ as a formula instead of a term. We will write $\overline{k} = \bigcup_{c\in\Delta} \overline{k}_c$ for the vector of all template parameters.

Applying templates. Let $\mathbf{k}_c \in \mathbb{D}^{|\overline{k}_c|}$ be concrete values for the template parameters $\overline{k}_c$ of component $c \in \Delta$. Furthermore, let $\mathbf{k} = \bigcup_{c\in\Delta} \mathbf{k}_c$ be the concrete values for *all* parameters $\overline{k}$. We write $P' = \mathsf{apply}(\mathbf{k}, P)$ to denote that program P is transformed to program P' by replacing all components $c \in \Delta$ with the expression $T_c(\mathbf{k}_c, \mathrm{Vars}(c))$. That is, in all templates, parameters $\overline{k}_c$ are replaced by the values defined in $\mathbf{k}_c$ and program variable symbols $\overline{v}_c$ are replaced by the respective variable names $\mathrm{Vars}(c)$. Finally, each component $c \in \Delta$ of P is textually replaced by the so instantiated template $T_c(\mathbf{k}_c, \mathrm{Vars}(c))$.

Reasoning about correctness. In order to check if a certain template instantiation yields a correctly repaired program, we define a formula

$$\begin{aligned}\mathsf{Rep}(\overline{i}, \overline{k}) =& \exists \overline{r} : \mathsf{Correct}(\overline{i}, \overline{r}) \wedge \bigwedge_{r\in\overline{r}_{\mathbb{D}}\setminus R_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r) \wedge \bigwedge_{r\in\overline{r}_{\mathbb{B}}\setminus R_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r) \wedge \\ & \bigwedge_{r\in R_{\mathbb{D}}} \exists \overline{v}_c : \overline{v}_c = \mathrm{Vals}(r) \wedge r = T_c(\overline{k}_c, \overline{v}_c) \wedge \\ & \bigwedge_{r\in R_{\mathbb{B}}} \exists \overline{v}_c : \overline{v}_c = \mathrm{Vals}(r) \wedge r \leftrightarrow T_c(\overline{k}_c, \overline{v}_c)\end{aligned} \tag{4.6}$$

where $R_{\mathbb{D}} = \{r \in \overline{r}_{\mathbb{D}} \mid \mathrm{CmpOf}(r) \in \Delta\}$, $R_{\mathbb{B}} = \{r \in \overline{r}_{\mathbb{B}} \mid \mathrm{CmpOf}(r) \in \Delta\}$ and c is short for $\mathrm{CmpOf}(r)$. The intuition behind Equation 4.6 is as follows. $\mathsf{Correct}(\overline{i}, \overline{r})$ expresses when the preprocessed program $\hat{P}$ satisfies the specification φ, depending on the unknown inputs $\overline{i}$ and the unknown values $\overline{r}$ returned by the components. Every symbol r produced by a correct component $c \notin \Delta$ is bound to the value $\mathrm{Org}(r)$ that would have been produced by the unmodified component c. Moreover, every symbol r that has been produced by an incorrect component $c \in \Delta$ is bound to the value that would be produced by the corresponding template T_c. This value is obtained by binding the symbols $\overline{v}_c$ to the values $\mathrm{Vals}(r)$ the program variables had when component c was executed to produce r (the equality is meant element-wise). Note that this seamlessly handles the case where a faulty component is executed multiple times: the same component is always associated with the same template and the same template parameters but the values of the program variables in the template can be different.

Lemma 35. *Let P be an incorrect program, $\mathbf{i} \in \mathbb{D}^k$ be an input vector, and $\mathbf{k} \in \mathbb{D}^{|\overline{k}|}$ be template parameter values. If $\mathcal{D}$ is precise (Definition 24), then* $\mathsf{Rep}(\mathbf{i}, \mathbf{k})$ *is* true *iff the program* $P' = \mathsf{apply}(\mathbf{k}, P)$ *fulfills the specification φ when executed with input* $\mathbf{i}$.

Proof. Let $\mathbf{r}$ be the concrete component results returned by calls to `cmpa` and `cmpc` in the preprocessed version $\hat{P}$ of P. According to Lemma 26, $\mathsf{Correct}(\mathbf{i}, \mathbf{r})$ evaluates to true iff $\hat{P}$ fulfills φ for input $\mathbf{i}$ and component results $\mathbf{r}$. The additional conjuncts in Equation 4.6 make Rep evaluate to true iff a special version $\hat{P}'$ of $\hat{P}$ satisfies φ for input $\mathbf{i}$. In $\hat{P}'$, all components $c \notin \Delta$ return the same value as the original implementation of c in P. All components $c \in \Delta$ return the values that would have been returned by template T_c, instantiated with parameter values defined in $\mathbf{k}$. This program $\hat{P}'$ behaves exactly as $P' = \mathsf{apply}(\mathbf{k}, P)$. □

Theorem 36. *Let* $\mathbf{k}$ *be a vector of concrete template parameter values such that* $\mathsf{Rep}(\mathbf{i}, \mathbf{k})$ *holds for all input vectors* $\mathbf{i} \in \mathbb{D}^k$. *If* $\mathcal{D}$ *is precise (Definition 24), then* $P' = \mathsf{apply}(\mathbf{k}, P)$ *is a correct program.*

Proof. Lemma 35 implies that P' cannot violate specification φ for any input. Hence, P' is correct. □

4.5.2 Computation of Repairs

Theorem 36 states that a repair can be computed as a satisfying assignment $\mathbf{k}$ for the variables $\overline{k}$ in the formula $\exists \overline{k} : \forall \overline{i} : \mathsf{Rep}(\overline{i}, \overline{k})$. This section discusses how such a satisfying assignment can be computed using an SMT solver. We avoid dealing with the quantifier alternation explicitly. Instead we apply Counterexample-Guided Inductive Synthesis (CEGIS) [197, 196], as defined in Algorithm 2.5 in Section 2.7, where a solution candidate is iteratively refined based on counterexamples. A user-defined bound on the maximum number of refinements is set to ensure termination.

Applying CEGIS via variable substitutions. Note that the formula $\mathsf{Rep}(\overline{i}, \overline{k})$, defined in Equation 4.6, contains existential quantifiers. However, these quantifiers do not encode any choice but are only introduced to increase readability: The symbols $\overline{\mathrm{v}}_c$ are uniquely defined by $\mathrm{Vals}(r)$, and thus act as abbreviations for the terms listed in $\mathrm{Vals}(r)$. Similarly, all repair symbols $r \in \overline{r}$ are uniquely defined either by $\mathrm{Org}_{\mathbb{D}}$, $\mathrm{Org}_{\mathbb{B}}$ or by a template T_c and can thus be seen as a shortcut for some term or formula as well.[4] A simple way for dealing with the existentially quantified symbols in Equation 4.6 is therefore to textually replace them with the term or formula they abbreviate. This gives a quantifier-free formula that only contains the symbols $\overline{i}$ and $\overline{k}$, so Algorithm 2.5 can be applied directly.

Applying CEGIS via variable renaming. Another approach is to rename the symbols $\overline{\mathrm{v}}_c$ to $\overline{\mathrm{v}}_{cr}$ for each $r \in R_{\mathbb{D}} \cup R_{\mathbb{B}}$ and remove the existential quantifiers in Equation 4.6 to obtain the formula

$$\begin{aligned}\mathsf{Rep}'(\overline{i}, \overline{k}, \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots) = {}& \mathsf{Correct}(\overline{i}, \overline{r}) \wedge \bigwedge_{r \in \overline{r}_{\mathbb{D}} \setminus R_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r) \wedge \bigwedge_{r \in \overline{r}_{\mathbb{B}} \setminus R_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r) \wedge \\ & \bigwedge_{r \in R_{\mathbb{D}}} \overline{\mathrm{v}}_{cr} = \mathrm{Vals}(r) \wedge r = T_c(\overline{k}_c, \overline{\mathrm{v}}_{cr}) \wedge \\ & \bigwedge_{r \in R_{\mathbb{B}}} \overline{\mathrm{v}}_{cr} = \mathrm{Vals}(r) \wedge r \leftrightarrow T_c(\overline{k}_c, \overline{\mathrm{v}}_{cr}).\end{aligned}$$

We have that $\mathsf{Rep}(\overline{i}, \overline{k}) \leftrightarrow \exists \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots : \mathsf{Rep}'(\overline{i}, \overline{k}, \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots)$, so we can compute satisfying assignments for Rep as satisfying assignments for Rep'. The CEGIS algorithm also needs to compute satisfying assignments of $\neg\, \mathsf{Rep}(\overline{i}, \mathbf{k})$ for some fixed values $\mathbf{k}$ of $\overline{k}$. This can be realized by defining

$$\begin{aligned}\mathsf{NegRep}'(\overline{i}, \overline{k}, \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots) = {}& \big(\neg\, \mathsf{Correct}(\overline{i}, \overline{r})\big) \wedge \bigwedge_{r \in \overline{r}_{\mathbb{D}} \setminus R_{\mathbb{D}}} r = \mathrm{Org}_{\mathbb{D}}(r) \wedge \bigwedge_{r \in \overline{r}_{\mathbb{B}} \setminus R_{\mathbb{B}}} r \leftrightarrow \mathrm{Org}_{\mathbb{B}}(r) \wedge \\ & \bigwedge_{r \in R_{\mathbb{D}}} \overline{\mathrm{v}}_{cr} = \mathrm{Vals}(r) \wedge r = T_c(\overline{k}_c, \overline{\mathrm{v}}_{cr}) \wedge \\ & \bigwedge_{r \in R_{\mathbb{B}}} \overline{\mathrm{v}}_{cr} = \mathrm{Vals}(r) \wedge r \leftrightarrow T_c(\overline{k}_c, \overline{\mathrm{v}}_{cr})\end{aligned}$$

and computing satisfying assignments to $\mathsf{NegRep}'(\overline{i}, \mathbf{k}, \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots)$. This works because of the duality $\neg\, \mathsf{Rep}(\overline{i}, \overline{k}) \leftrightarrow \exists \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots : \mathsf{NegRep}'(\overline{i}, \overline{k}, \overline{r}, \overline{\mathrm{v}}_{cr_1}, \overline{\mathrm{v}}_{cr_2}, \ldots)$. Using Rep' and NegRep', the CegisRepair procedure from Algorithm 4.2 can be applied. It differs from Algorithm 2.5 only in using Rep' instead of F, NegRep' instead of $\neg F$ and some more renaming of the auxiliary variables.

[4] Note that $\mathrm{Org}_{\mathbb{D}}(r)$, $\mathrm{Org}_{\mathbb{B}}(r)$, $\mathrm{Vars}(r)$ and thus also $T_c(\overline{k}_c, \overline{\mathrm{v}}_c)$ may contain symbols from $\overline{r}$. Still, there cannot be any circular dependencies because $\mathrm{Org}_{\mathbb{D}}(r)$, $\mathrm{Org}_{\mathbb{B}}(r)$, $\mathrm{Vars}(r)$ can only contain $\overline{r}$-symbols that occur *before* r in the vector $\overline{r}$. This order among the symbols in $\overline{r}$ is imposed by the order in which the symbols are created during symbolic or concolic execution.

Algorithm 4.2 CEGISREPAIR: The CEGIS algorithm applied to computing program repairs.

```
 1: procedure CEGISREPAIR($\mathsf{Rep}'(\overline{i},\overline{k},\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2},,\ldots), \mathsf{NegRep}'(\overline{i},\overline{k},\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2}\ldots)$),
 2: returns: An assignment $\mathbf{k}$ for $\overline{k}$ such that $\forall\overline{i}:\exists\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2},\ldots:\mathsf{Rep}'(\overline{i},\mathbf{k},\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2},\ldots)$ or "fail"
 3:     $G(\overline{k},\ldots) := \mathsf{true}$
 4:     while true do
 5:         $(\mathsf{sat},\mathbf{k}) :=$ SMTSATMODEL($G(\overline{k},\ldots)$)
 6:         if $\mathsf{sat} = \mathsf{false}$ then
 7:             return "fail"
 8:         end if
 9:         $(\mathsf{sat},\mathbf{i}) :=$ SMTSATMODEL($\mathsf{NegRep}'(\overline{i},\mathbf{k},\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2},\ldots)$)
10:         if $\mathsf{sat} = \mathsf{false}$ then
11:             return $\mathbf{k}$
12:         end if
13:         $\overline{r}',\overline{\mathrm{v}}'_{cr_1},\overline{\mathrm{v}}'_{cr_2},\ldots := \mathsf{createFreshCopies}(\overline{r},\overline{\mathrm{v}}_{cr_1},\overline{\mathrm{v}}_{cr_2},\ldots)$
14:         $G(\overline{k},\ldots) := G(\overline{k},\ldots) \wedge \mathsf{Rep}'(\mathbf{i},\overline{k},\overline{r}',\overline{\mathrm{v}}'_{cr_1},\overline{\mathrm{v}}'_{cr_2}\ldots)$
15:     end while
16: end procedure
```

Example 37. We continue Example 22 with the diagnostic information $\mathcal{D}$ from Example 25. For the diagnosis $\Delta = \{c_3\}$ and the template $T_{c_3}(\overline{k},\overline{\mathrm{v}}) = k_0 + k_1 \cdot \mathrm{v}_1 + k_2 \cdot \mathrm{v}_2 + k_3 \cdot \mathrm{v}_3$ we obtain

$$\begin{aligned}
\mathsf{Rep}\big((\alpha,\beta),(k_0,k_1,k_2,k_3)\big) =& \exists\rho_1,\rho_2,\rho_3:\big((\rho_2 \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee ((\neg\rho_2) \wedge \rho_1 \geq \alpha \wedge \rho_1 \geq \beta)\big) \wedge \\
& \qquad\qquad\qquad (\rho_1 = \alpha) \wedge \big(\rho_2 = (\beta > \alpha)\big) \wedge \\
& \qquad\qquad\qquad \exists\mathrm{v}_1,\mathrm{v}_2,\mathrm{v}_3:(\mathrm{v}_1 = \alpha) \wedge (\mathrm{v}_2 = \beta) \wedge (\mathrm{v}_3 = \rho_1) \wedge \\
& \qquad\qquad\qquad\qquad\qquad (\rho_3 = k_0 + k_1 \cdot \mathrm{v}_1 + k_2 \cdot \mathrm{v}_2 + k_3 \cdot \mathrm{v}_3) \\
=& \exists\rho_3:\big((\beta > \alpha \wedge \rho_3 \geq \alpha \wedge \rho_3 \geq \beta) \vee (\beta \leq \alpha \wedge \alpha \geq \alpha \wedge \alpha \geq \beta)\big) \wedge \\
& \qquad (\rho_3 = k_0 + k_1 \cdot \alpha + k_2 \cdot \beta + k_3 \cdot \alpha) \\
=& (k_0 + k_1 \cdot \alpha + k_2 \cdot \beta + k_3 \cdot \alpha \geq \beta) \vee (\beta \leq \alpha)
\end{aligned}$$

The first candidate solution in CEGIS is arbitrary because it is computed as a satisfying assignment for the formula true. We could obtain $\mathbf{k}_0 = (0,0,0,0)$, which corresponds to replacing component c_3 in program P by the expression $T_{c_3}\big(\mathbf{k}_0,(\mathtt{a},\mathtt{b},\mathtt{r})\big) =$ "`0`". The next step is to compute a counterexample $\mathbf{i}_0$ to the correctness of $\mathbf{k}_0$ as a satisfying assignment for $\neg\,\mathsf{Rep}(\overline{i},\mathbf{k}_0) = \neg\big((0 \geq \beta) \vee (\beta \leq \alpha)\big)$. This formula is satisfiable, so $\mathbf{k}_0$ is not a correct repair. We could get $\mathbf{i}_0 = (0,10)$ as a counterexample. Next, an improved candidate $\mathbf{k}_1$ is computed as a satisfying assignment to $\mathsf{Rep}(\mathbf{i}_0,\overline{k}) = (k_0 + k_2 \cdot 10 \geq 10)$. We may get $\mathbf{k}_1 = (10,0,0,0)$, which corresponds to replacing c_3 with "`10`". Again, we find a counterexample $\mathbf{i}_1 = (1,100)$ disproving $\mathbf{k}_1$ as a satisfying assignment for $\neg\,\mathsf{Rep}(\overline{i},\mathbf{k}_1) = \neg\big((10 \geq \beta) \vee (\beta \leq \alpha)\big)$. The next candidate is computed as satisfying assignment $\mathbf{k}_2$ for $\mathsf{Rep}(\mathbf{i}_0,\overline{k}) \wedge \mathsf{Rep}(\mathbf{i}_1,\overline{k}) = (k_0+k_2 \cdot 10 \geq 10) \wedge (k_0+k_1+k_2 \cdot 100+k_3 \geq 100)$. Assume that the SMT solver returns $\mathbf{k}_2 = (1,0,1,0)$, which corresponds to the expression "`1 + b`". Since $\neg\,\mathsf{Rep}(\overline{i},\mathbf{k}_2) = \neg\big((1+\beta \geq \beta) \vee (\beta \leq \alpha)\big)$ is unsatisfiable, our procedure terminates, suggesting to replace c_3 in P by "`1 + b`" as a repair.

Computing multiple repairs. As illustrated by the result of Example 37, a repaired program P' may satisfy the specification φ, but may still fail to satisfy the user. One possible reason is that the specification is incomplete. The repair may also be correct in a functional sense but suboptimal or undesirable in some other respect. With our approach, we can easily compute multiple different repairs and let the user choose one: when a repair $\mathbf{k}$ is found in Line 10 of Algorithm 4.2, we store it, update $G(\overline{k},\ldots) := G(\overline{k},\ldots) \wedge \overline{k} \neq \mathbf{k}$ with constraints that require satisfying assignments for G to be different from the found solution $\mathbf{k}$ (in at least one parameter value), and continue to execute the loop in order to find additional repairs. In the same way, additional constraints for repairs could also be imposed by the user. For instance, for the template from Example 37, the constraint $k_1 = 0 \vee k_3 = 0$ added to G can express that the synthesized expression cannot refer to both variables `a` and `r` simultaneously.

4.5.3 Heuristics

In this section, we point out some issues with the repair synthesis process from the previous section and propose heuristics to address them. Computing repairs with CEGIS can be seen as a game between two players. Player 1 comes up with candidates and Player 2 attempts to disprove them. Player 1 wins if she manages to find a correct candidate. Player 2 wins if she has ruled out every possible candidate. In our experiments, we discovered two issues of this procedure.

Issue 1: Candidates may diverge. Even if simple repairs exist, the play between Player 1 and Player 2 may end up computing and excluding more and more complex candidates. For instance, for one program in our experiments, the sequence of candidates

Iteration	Candidate replacement for a certain component
0	`0`
1	`-v0`
2	`250*v1 + 248*v2 - 2*v3 - v4`
3	and so on, becoming more and more complex

was observed, although the constant `500` was a repair for that component. That is, the sequence of repair candidates may diverge, with the search getting lost in vastness of the candidate space.

Issue 2: Progress may be insufficient. Even if candidates remain simple, the progress may still be insufficient if both player do the least to fulfill their duty. For instance, for Example 37, the CEGIS approach could produce the following sequence of candidates:

Iteration	Candidate replacement for c_3	Counterexample $\overline{i}$
0	`0`	$(0, 1)$
1	`1`	$(1, 2)$
2	`2`	$(2, 3)$
3	`3`	$(3, 4)$
4	and so on	and so on

The constant `0` as a replacement for component c_3 is not high enough for the program input $\alpha = 0$ and $\beta = 1$. The constant `1` works for this counterexample, but is not high enough for input $(1, 2)$. The constant `2` works for the first two counterexamples, but is not high enough for $(2, 3)$, etc. Depending on the size of $\mathbb{D}$, this interaction may not terminate (in reasonable time). Note that each counterexample subsumes all the previous ones. Hence, if Player 2 would have started with the counterexample $\overline{i} = (100, 101)$, the first 100 iterations would have been saved.

Solution. We solve these two issues heuristically by improving the two players. Intuitively, we want "simple" candidates and "nasty" counterexamples.

Computing simple candidates. We say that a candidate is "simple" if many template parameters k_i are small or, even better, equal to some special value d_i that makes terms in the template disappear (e.g., zero in case of a template for linear expressions). To implement this idea, we define a set $\mathcal{S}$ of constraints $S(\overline{k}) \in \mathbb{F}$ over the template parameters $\overline{k}$ as

$$\mathcal{S} = \bigcup_{k_i \in \overline{k}} (k_i = d_i) \cup (-s \leq k_i \leq s),$$

where s is a constant defining what "small" means. In Algorithm 4.2, we can now compute simple candidates by modifying Line 5 to compute not just any satisfying assignment for $G(\overline{k}, \dots)$ but one that also satisfies as many constraints $S(\overline{k}) \in \mathcal{S}$ as possible. This problem is known as the (partial) Maximum Satisfiability Modulo Theories (MAX-SMT) problem [162] and supported natively by several SMT solvers. An efficient MAX-SMT engine can also be built on top of any SMT solver that supports the computation of unsatisfiable cores [94].[5]

Computing nasty counterexamples. We say that a counterexample is “nasty” if it contains large, uncorrelated values. Again, we define a set $\mathcal{N}$ of constraints $N(\overline{i}) \in \mathbb{F}$ over the input symbols $\overline{i}$ as

$$\mathcal{N} = \bigcup_{i \in \overline{i}} (i \geq l \vee i \leq -l),$$

where l is a constant that is much larger than s. Similar as before, we change Line 9 in Algorithm 4.2 to a MAX-SMT computation in order to compute a counterexample $\mathbf{i}$ that satisfies as many constraints from $\mathcal{N}$ as possible. In order to break correlations between values in the counterexample we additionally randomize it: values are changed to large random values as long as the modified input vector is still a counterexample.

4.6 Using Test Cases as a Specification

Natively, our software repair flow only supports assertions in the code as specification. Assertions can also be used to compare the outcome of a program with that of a reference implementation. However, in some cases it can be difficult to come up with meaningful assertions. In this section, we therefore discuss how our debugging approach can be extended to using test cases as a specification. Intuitively, a test case defines the expected output of the program for some concrete input.

Programs and specifications. In order to model output in our program P, we introduce a special function `output(x)`. It takes one integer value `x` as parameter, which is considered to be an output of the program. This function can be called repeatedly (possibly interleaved with calls to `input`) if the program outputs a sequence of values. Our specification φ is now given as a set T of test cases t. Formally, we define a test case $t = (\mathbf{i}, \mathbf{o}) \in \mathbb{D}^* \times \mathbb{D}^*$ to consist of a vector $\mathbf{i}$ of concrete input values and a vector $\mathbf{o}$ of expected output values. Whenever a call to the special function `input` is encountered (and for all parameters of the entry function), the next value from $\mathbf{i}$ is read.[6] Whenever a call to `output(x)` is encountered, the value of `x` is compared with the next element in $\mathbf{o}$.[7] If the values differ, the program terminates and we say that the test case t *failed*. Otherwise, if the program terminates normally by reaching the end of the entry function, we say that the test case *passed*. The program P is said to *satisfy the specification* φ if all test cases $t \in T$ pass.

Combination with assertions and assumptions. Test cases can easily be combined with assertions and assumptions as a specification. Whenever the execution of a test case t encounters a statement `assume(c)` with `c` being false, the execution terminates and the test case t is marked as passed. Whenever some statement `assert(c)` with `c` $=$ false is executed, the execution terminates and t is marked as failed. That is, in contrast to Section 4.1.1, the assertions are only considered for the concrete input values defined by the test cases.

Program analysis. The program is preprocessed as usual (see Section 4.2.2). The preprocessed program $\hat{P}$ is then analyzed using symbolic or concolic execution for every test case t one after the other. This works essentially as explained in Section 4.2.3. However, calls to the special function `input` do not produce a new input symbol $i \in \overline{i}$ but always return a concrete value. Note that program variables can still have a symbolic value (as a function of $\overline{r}$) due to component executions. Calls to the function `output(x)` are treated as if they where a shortcut for `if(x != e) exitErr();`, where `e` is the expected output value read from the test case. That is, calls to `output` will, in general, make the

[5] Fu and Malik [94] describe the approach for the Boolean case but an extension from SAT to SMT is straightforward [162].

[6] After the length of $\mathbf{i}$ is exceeded, `input` returns arbitrary values.

[7] After the length of $\mathbf{o}$ is exceeded, `output` can be called with arbitrary values.

symbolic execution fork into a branch where the output is as expected and one where it is not. Let

$$\mathcal{D}^t = (\mathsf{CMP}, \mathrm{TypOf}, \mathrm{Vars}, \emptyset, \overline{r}^t_{\mathbb{D}}, \overline{r}^t_{\mathbb{B}}, \mathrm{CmpOf}^t, \mathrm{Org}^t_{\mathbb{D}}, \mathrm{Org}^t_{\mathbb{B}}, \mathrm{Vals}^t, \mathsf{Correct}^t(\overline{r}^t))$$

with $\overline{r}^t = \overline{r}^t_{\mathbb{D}} \cup \overline{r}^t_{\mathbb{B}}$ be the so obtained diagnostic information for test case t. Note that the vector of input symbols is empty. Assuming that all symbol vectors $\overline{r}^t$ are disjoint[8], we compute the combined diagnostic information $\mathcal{D} = (\mathsf{CMP}, \mathrm{TypOf}, \mathrm{Vars}, \emptyset, \overline{r}_{\mathbb{D}}, \overline{r}_{\mathbb{B}}, \mathrm{CmpOf}, \mathrm{Org}_{\mathbb{D}}, \mathrm{Org}_{\mathbb{B}}, \mathrm{Vals}, \mathsf{Correct}(\overline{r}))$ with

$$\begin{aligned}
\overline{r}_{\mathbb{D}} &= \bigcup_{t\in T} \overline{r}^t_{\mathbb{D}}, & \mathrm{Org}_{\mathbb{D}}(r) &= \mathrm{Org}^t_{\mathbb{D}}(r) \text{ if } r \in \overline{r}^t, & \mathrm{CmpOf}(r) &= \mathrm{CmpOf}^t(r) \text{ if } r \in \overline{r}^t,\\
\overline{r}_{\mathbb{B}} &= \bigcup_{t\in T} \overline{r}^t_{\mathbb{B}}, & \mathrm{Org}_{\mathbb{B}}(r) &= \mathrm{Org}^t_{\mathbb{B}}(r) \text{ if } r \in \overline{r}^t, & \mathrm{Vals}(r) &= \mathrm{Vals}^t(r) \text{ if } r \in \overline{r}^t,\\
\overline{r} &= \overline{r}_{\mathbb{D}} \cup \overline{r}_{\mathbb{B}} & \mathsf{Correct}(\overline{r}) &= \bigwedge_{t\in T} \mathsf{Correct}^t(\overline{r}^t)
\end{aligned}$$

Fault localization. In principle, the SMT-based fault localization approach presented in Section 4.3 can be applied without any modifications when using diagnostic information obtained with test cases T as a specification. However, since the set of input symbols $\overline{\imath}$ is empty, some simplifications can be made. The universal quantification in repairability checks according to Equation 4.3 falls away, so there is no need to approximate this universal quantification with a finite conjunction (as done in Equation 4.4). However, an approximation in the flavor of Equation 4.4 can still be performed by considering only a subset of the test cases when constructing $\mathsf{Correct}(\overline{r})$ from the individual conjuncts $\mathsf{Correct}^t(\overline{r}^t)$. The reason is that each conjunct $\mathsf{Correct}^t(\overline{r}^t)$ expresses program correctness for one concrete vector $\mathbf{i}$ of inputs (namely the one in test case $t = (\mathbf{i}, \mathbf{o})$). Hence, $\mathsf{Correct}^t(\overline{r}^t)$ can be seen as an instantiation $\mathsf{Correct}'(\mathbf{i}, \overline{r})$ of a hypothetical formula $\mathsf{Correct}'(\overline{\imath}, \overline{r})$ that captures the program correctness for all inputs.

Repair. The CEGIS-based program repair approach presented in Section 4.5 can also be simplified significantly if the vector of input symbols $\overline{\imath}$ is empty. In fact, a correct template instantiation can be computed with a single SMT solver call as a satisfying assignment to $\mathsf{Rep}(\overline{k})$, as defined by Equation 4.6 with $\overline{\imath} = \emptyset$. However, depending on the program size and the number of test cases, the correctness formula $\mathsf{Correct}(\overline{r})$ used to build $\mathsf{Rep}(\overline{k})$ may be rather large and, consequently, expensive to solve. It can thus still make sense to follow an approach of iterative refinements: By including only the conjuncts $\mathsf{Correct}^t(\overline{r}^t)$ with $t \in T'$ for some $T' \subset T$ in $\mathsf{Correct}(\overline{r})$, we can compute a repair candidate that is guaranteed to pass at least the subset T' of the test cases T. If the candidate fails on some test $t \in T \setminus T'$, then T' can be enlarged (at least by t). The following section generalizes this principle of refining the correctness formula on demand.

4.7 Repair Synthesis with On-The-Fly Program Analysis

Section 4.5 already presented a basic solution for synthesizing repairs using templates. This section discusses some issues of this solution and then presents an improved synthesis flow addressing these issues. The section concludes with a discussion of the main benefits of the improved flow.

Recall that the template-based CEGIS approach presented in Section 4.5 synthesizes repairs for incorrect programs by refining candidates iteratively based on counterexamples. It relies on the availability of a monolithic correctness formula $\mathsf{Rep}(\overline{\imath}, \overline{k})$ that expresses which repair candidates (values for the template parameters $\overline{k}$) would be correct for which inputs (values for the variables $\overline{\imath}$). This one formula is used both for computing candidates and for refuting them. Candidates are computed as satisfying assignments for the variables $\overline{k}$ with fixed values for $\overline{\imath}$. Counterexamples disproving candidates are computed as assignments to $\overline{\imath}$ with fixed values for $\overline{k}$.

Issues. While working with one monolithic correctness formula $\mathsf{Rep}(\overline{\imath}, \overline{k})$ has a certain elegance, it also comes with some issues. First, exhaustive program analysis to build a perfectly accurate correctness

[8] Otherwise, they can easily be renamed.

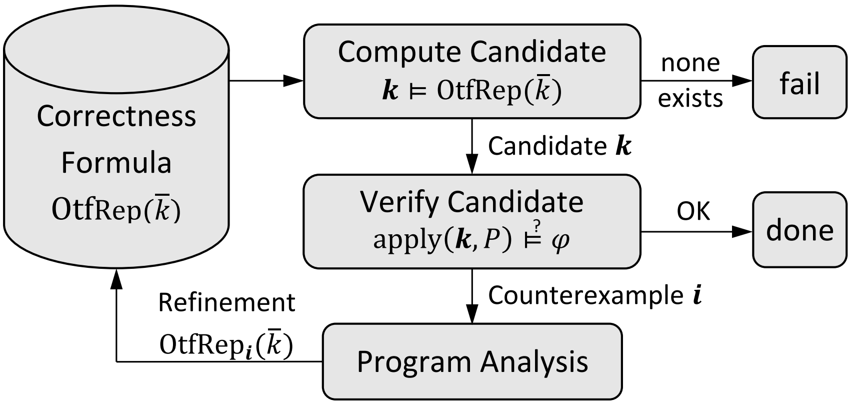

Correctness
Formula
OtfRep($\bar{k}$)
Compute Candidate
$\boldsymbol{k} \vDash$ OtfRep($\bar{k}$)
none exists
fail
Candidate $\boldsymbol{k}$
Verify Candidate
apply($\boldsymbol{k}, P$) $\overset{?}{\vDash} \varphi$
OK
done
Counterexample $\boldsymbol{i}$
Refinement
OtfRep$_{\boldsymbol{i}}$($\bar{k}$)
Program Analysis

Figure 4.2: Overview of our repair method with on-the-fly program analysis. A repair candidate is computed as a satisfying assignment $\mathbf{k}$ to a correctness formula $\mathsf{OtfRep}(\overline{k})$, which is initially true. The candidate program $P' = \mathsf{apply}(\mathbf{k}, P)$ is then verified by some means. If a counterexample $\mathbf{i}$ is found, a program analysis step refines $\mathsf{OtfRep}(\overline{k})$ with constraints $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k})$ on the parameters $\overline{k}$ that ensure correctness for input $\mathbf{i}$. This loop is repeated until a correct repair is found or $\mathsf{OtfRep}(\overline{k})$ becomes unsatisfiable.

formula is often infeasible, especially if the program contains loops or recursive functions. Yet, when setting bounds on the analysis depth (e.g., by limiting the number of loop unrollings or considered execution paths) one runs the risk of abstracting away the wrong information. Second, even if exhaustive program analysis is feasible, it may take long and produce an unnecessarily large correctness formula. After all, for computing repair candidates using CEGIS, the correctness formula needs to be accurate for some (typically few) inputs only. In this light, constructing a formula that is accurate for *all* inputs can be wasteful. Third, computing counterexamples as satisfying assignments to the negated correctness formula may not be ideal. Depending on the program and its specification formalism, other techniques such as test case execution or invoking a specialized and optimized model checker may be a better option.

Concept. We remedy these issues in our improved repair synthesis flow by doing program analysis on the fly during the repair process, thereby analyzing the program only for the counterexamples that have been found. Furthermore, we decouple the computation of repair candidates from their verification. This allows us to use various verification techniques and specification formats.

4.7.1 Solution

The input to our repair method with on-the-fly program analysis is an incorrect program P, a specification φ, and a diagnosis $\Delta \subseteq \mathsf{CMP}$. In contrast to Section 4.5, no diagnostic information $\mathcal{D}$ is required because the program will be analyzed on demand. We do, however, use templates as in Section 4.5. That is, for every potentially incorrect program component $c \in \Delta$, we construct a template $T_c(\overline{k}_c, \overline{v}_c)$ for a new implementation of the component. As before, the vector of all template parameters is denoted by $\overline{k} = \bigcup_{c \in \Delta} \overline{k}_c$. Applying a template instantiation $\mathbf{k}$ to the program P is again denoted by $P' = \mathsf{apply}(\mathbf{k}, P)$.

Overview. Figure 4.2 outlines our approach to compute a repair by finding an appropriate template instantiation with on-the-fly program analysis. We maintain a quantifier-free correctness formula $\mathsf{OtfRep}(\overline{k}) \in \mathbb{F}$ over the template parameters $\overline{k}$. Intuitively, this correctness formula expresses (an incomplete set of) constraints that must be satisfied by a template instantiation $\mathbf{k}$ to yield a correct repair $P' = \mathsf{apply}(\mathbf{k}, P)$. The correctness formula is of the form

$$\mathsf{OtfRep}(\overline{k}) = \bigwedge_{\mathbf{i}} \mathsf{OtfRep}_{\mathbf{i}}(\overline{k}),$$

where each $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k})$ is a quantifier-free formula that expresses constraints that must be satisfied by a template instantiation to be correct for input $\mathbf{i}$. Initially, $\mathsf{OtfRep}(\overline{k})$ is true. As a first step in a loop, we compute a repair candidate $\mathbf{k}$ as a satisfying assignment for the formula $\mathsf{OtfRep}(\overline{k})$. If $\mathsf{OtfRep}(\overline{k})$ is unsatisfiable, the loop aborts, signaling that no repair with this template exists. Otherwise, a candidate program $P' = \mathsf{apply}(\mathbf{k}, P)$ is constructed. Next, we verify if P' satisfies the specification φ. If this is the case, P' is reported as a repair and the loop terminates. Otherwise, the verification step returns a counterexample $\mathbf{i}$, which is an input for which P' violates φ. Using this counterexample $\mathbf{i}$, we perform a program analysis step in order to infer additional constraints $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k})$ that ensure correctness for input $\mathbf{i}$. These additional constraints are conjoined with $\mathsf{OtfRep}(\overline{k})$ and the next iteration starts. In this way, our method keeps analyzing the program for more and more inputs, and improving the repair candidates until a correct one is found. A user-defined bound on the maximum number of iterations can be imposed to ensure termination. The next subsections explain the individual steps in more detail.

4.7.1.1 Repair Candidate Computation

Since $\mathsf{OtfRep}(\overline{k})$ is a (quantifier free) formula from $\mathbb{F}$, a satisfying assignment can be computed with one SMT solver call $(\mathsf{sat}, \mathbf{k}) := \textsc{SmtSatModel}\big(\mathsf{OtfRep}(\overline{k})\big)$. Alternatively, the heuristics from Section 4.5.3 for computing simple repair candidates can be applied by solving a MAX-SMT problem.

4.7.1.2 Repair Candidate Verification

The verification of repair candidates can be performed in many ways. The prerequisite is that the verification method is able to produce a counterexample in case of incorrectness. This increased flexibility in candidate verification also comes with increased flexibility in the specification mechanism.

Symbolic or concolic execution. When using assertions in the code as specification φ, we can apply (standard) symbolic or concolic execution to verify a candidate program P'. Since we are not interested in a correctness formula but only in one counterexample, the analysis can be stopped as soon as the first execution path that violates the specification is encountered. In the case of symbolic execution, a satisfying assignment to the respective path condition is then computed to obtain the counterexample $\mathbf{i}$. When using concolic execution, a concrete input vector to activate the execution path is available anyway.

Model checking. Another possibility is to use a software model checker taking assertions in the code as a specification. This includes bounded software model checkers such as CBMC [61] or abstraction-based tools such as BLAST [20] or SATABS [62]. Model checkers are typically able to prove incorrectness by giving a counterexample in the form of an input vector for which an assertion is violated. There is also a competition, called SV-COMP[9], among tools for software verification. Since 2015, tools are also required to produce verifiable counterexamples in this competition [19].

Test case execution. When using test cases as specification, the natural way to verify candidates is to execute them. As in Section 4.6, we consider test cases $t = (\mathbf{i}, \mathbf{o})$ that are given as an input vector $\mathbf{i}$ together with a vector $\mathbf{o}$ of expected outputs. A counterexample is the input part $\mathbf{i}$ of a failing test case.

Verification by the user. Repair candidates can even be verified by a human user. Here, a counterexample is again an input vector. The user also defines the corresponding expected output, which serves as a specification. That is, no explicit specification needs to be available initially. The user simply creates a specification in the form of input-output examples (which are essentially test cases) on demand. The repair engine learns a fix of the incorrect program using the input-output examples given by the user in response to the candidates. Constructing input-output examples to rule out the *presented* candidates can be significantly less work than creating a test suite that rules out *all* undesirable candidates before starting the repair engine.

Notation. We will write $(\mathsf{OK}, \mathbf{i}) := \textsc{Verify}(P', \varphi)$ to denote the verification of the candidate program P' with respect to the specification φ using some method. If P' satisfies φ, then OK is set to

[9] `http://sv-comp.sosy-lab.org` (last visit on 2015-08-01).

true, otherwise to false. If $\mathsf{OK} = \mathsf{false}$, then $\mathbf{i} \in \mathbb{D}^*$ contains a counterexample, which is a vector of concrete input values for which P' violates φ.

4.7.1.3 On-The-Fly Program Analysis

The crucial step of our improved repair approach is program analysis, which is deliberately incomplete: We only look at behavior that is possible under one particular input assignment, namely the counterexample $\mathbf{i}$ found in the preceding verification step. Similar to Section 4.6, we do so using a customized version of symbolic or concolic execution.

Preprocessing. Recall that we have introduced a template $T_c(\overline{k}_c, \overline{\mathtt{v}}_c)$ for every potentially incorrect program component $c \in \Delta$. The symbols $\overline{k}_c$ are template parameters and $\overline{k} = \bigcup_{c \in \Delta} \overline{k}_c$ is their union. The symbols $\overline{\mathtt{v}}_c$ represent the values of the variables in scope when component c is executed. We now introduce a fresh (global) program variable $\mathtt{k}$ for every template parameter $k \in \overline{k}$ and construct the program $\hat{P}' = \mathsf{apply}(\overline{\mathtt{k}}, P)$, in which all components $c \in \Delta$ are replaced with the program expression $T_c\big(\overline{\mathtt{k}}_c, \mathrm{Vars}(c)\big)$. That is, in the template $T_c(\overline{k}_c, \overline{\mathtt{v}}_c)$, we replace all parameter symbols $\overline{k}_c$ with the corresponding parameter variables $\overline{\mathtt{k}}_c$ and all variable symbols $\overline{\mathtt{v}}_c$ with the names of the variables $\mathrm{Vars}(c)$ that are in scope when component c is executed. Each program component $c \in \Delta$ is then replaced with the so constructed program expression $T_c\big(\overline{\mathtt{k}}_c, \mathrm{Vars}(c)\big)$.

Example 38. Consider the repair problem from Example 37 again.

```
1 int max(int a, int b) {
2   int r = a;
3   if(b > a)
4     r = a + 2;
5   assert(r >= a && r >= b);
6   return r;
7 }
```

```
  int k0, k1, k2, k3;
1 int max(int a, int b) {
2   int r = a;
3   if(b > a)
4     r = k0 + k1*a + k2*b + k3*r;
5   assert(r >= a && r >= b);
6   return r;
7 }
```

The source code listing on the left shows the program P from Example 22. The listing on the right shows the preprocessed version $\hat{P}' = \mathsf{apply}(\overline{\mathtt{k}}, P)$ for the diagnosis $\Delta = \{c_3\}$ and the template $T_{c_3}(\overline{k}, \overline{\mathtt{v}}) = k_0 + k_1 \cdot \mathtt{v}_1 + k_2 \cdot \mathtt{v}_2 + k_3 \cdot \mathtt{v}_3$. With the program variables $\overline{\mathtt{k}} = (\mathtt{k0}, \mathtt{k1}, \mathtt{k2}, \mathtt{k3})$ to represent the template parameters $\overline{k} = (k_0, k_1, k_2, k_3)$ and $\mathrm{Vars}(c_3) = (\mathtt{a}, \mathtt{b}, \mathtt{r})$, we have that $T_{c_3}(\overline{\mathtt{k}}, \mathrm{Vars}(c_3)) =$ `k0 + k1*a + k2*b + k3*r`. This program expression replaces the component c_3 (the expression `a + 2` in Line 4) in the preprocessed program $\hat{P}'$ shown on the right.

Symbolic execution. We perform symbolic execution similar to Section 4.6. The program inputs are fixed to the values $\mathbf{i}$ in the counterexample. That is, for all parameters of the entry function and for all calls to the function `input`, the next concrete value from $\mathbf{i}$ is used. The symbols $\overline{k}$ are used as symbolic values of the template parameter variables $\overline{\mathtt{k}}$. Note that these program variables $\overline{\mathtt{k}}$ have an unknown but constant value, so their value is represented by the same symbols $\overline{k}$ throughout the entire program execution. Assertions and assumptions in the code are handled as usual (see Section 4.2.3.2). Calls to the function `output` are only considered if the specification φ is given as a set T of test cases. In this case, we search for a test case $t = (\mathbf{i}, \mathbf{o}) \in T$ for which the input part matches our counterexample. As in Section 4.6, calls to the function `output(x)` are then treated as if they were a shortcut for `if(x != e) exitErr();`, where `e` is the next expected output value read from the output part $\mathbf{o}$ of the test case t. In order to limit the effort for program analysis as well as the size of the resulting formulas, we set a user-defined bound on the maximum number and length of the execution paths to consider.

Computing a correctness formula. As in Section 4.2.3.2, we divide the leaf nodes of the symbolic execution tree into three sets: FAIL is the set of all nodes that are marked with a specification violation (i.e., ended in `exitErr()`), PASS is the set of all nodes where the program terminated normally (or in an assumption violation), and OPEN contains the remaining leaf nodes where the execution was aborted because some analysis depth limit was exceeded. Each path condition associated with a node n of the symbolic execution tree is a formula $\mathsf{PC}_n(\overline{k}) \in \mathbb{F}$ over the template parameters $\overline{k}$. We finally compute

$$\mathsf{OtfRep}_{\mathbf{i}}(\overline{k}) = \bigwedge_{n \in \mathsf{FAIL}} \neg \mathsf{PC}_n(\overline{k}). \tag{4.7}$$

We denote this computation by $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k}) := \textsc{Analyze}(\hat{P}', \mathbf{i})$ with $\hat{P}' = \mathsf{apply}(\overline{\mathrm{k}}, P)$.

Lemma 39. *Let* $\mathbf{k}$ *be values for the template parameters* $\overline{k}$. *Assume that our approach for on-the-fly program analysis using symbolic execution has been applied to the preprocessed program* $\hat{P}' = \mathsf{apply}(\overline{k}, P)$ *with counterexample* $\mathbf{i}$ *exhaustively (i.e.,* $\mathit{OPEN} = \emptyset$*) and without abstracting the program semantics. Then* $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ *is* true *iff* $P' = \mathsf{apply}(\mathbf{k}, P)$ *satisfies the specification* φ *for input* $\mathbf{i}$.

Proof. Every leaf node n in the symbolic execution tree represents an execution path through the program $\hat{P}'$. Let n_e be the node that represents the execution path p_e that is induced by the inputs $\mathbf{i}$ and parameter values $\mathbf{k}$. Since all leaf nodes have disjunct path conditions, $\mathsf{PC}_n(\mathbf{k})$ is true iff $n = n_e$. $\neg\mathsf{PC}_{n_e}(\mathbf{k}) = \mathsf{false}$ is a conjunct of $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ iff $n_e \in \mathsf{FAIL}$, i.e., iff p_e violates the specification. Hence, $\hat{P}'$ satisfies φ for input $\mathbf{i}$ and parameter values $\mathbf{k}$ iff $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ is true. The program P' differs from $\hat{P}'$ only in having the parameters fixed to $\mathbf{k}$. Consequently, P' also satisfies φ for input $\mathbf{i}$ iff $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ is true. □

Lemma 39 states that $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k}) = \mathsf{true}$ is both a necessary and a sufficient condition for $P' = \mathsf{apply}(\mathbf{k}, P)$ to satisfy the specification φ for input $\mathbf{i}$ (given that program analysis was applied without approximations). The formula $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k})$ is therefore conjoined to the correctness formula $\mathsf{OtfRep}(\overline{k})$ (see Figure 4.2) in order to exclude candidates that fail for the input vector $\mathbf{i}$ in the next iterations.

Concolic execution. Since concolic execution is merely a variant of symbolic execution, it can easily be applied instead of symbolic execution, as explained in Section 4.2.3.3. In the context of on-the-fly program analysis, the set of symbols $\overline{k}$ that can appear in path conditions is fixed. Hence, in contrast to Section 4.2.3.3, no special care in managing symbol uniqueness across multiple concolic execution runs is necessary.

4.7.1.4 Algorithm

The procedure RepairOtf in Algorithm 4.3 combines the different steps from the previous subsections into an algorithm, thereby formalizing their interplay as already outlined in Figure 4.2. Based on this algorithm, we will now work out correctness guarantees.

Theorem 40. *Assume that* $\textsc{Analyze}(\hat{P}', \mathbf{i})$ *always analyzes the correctness of program* $\hat{P}'$ *for input* $\mathbf{i}$ *exhaustively and without abstracting the program semantics. Then the procedure* RepairOtf *from Algorithm 4.3 will (a) only return "*fail*" if no repair with the given templates* $T_c(\overline{k}_c, \overline{v}_c)$ *satisfies the specification* φ*, (b) only return a repair* P' *if this repair satisfies* φ*, and (c) never investigate the same template parameters* $\mathbf{k}$ *twice.*

Proof. Statement (a) follows from Lemma 39: $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ is true iff $P' = \mathsf{apply}(\mathbf{k}, P)$ satisfies φ for input $\mathbf{i}$. That is, $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k}) = \mathsf{true}$ is a necessary condition for $P' = \mathsf{apply}(\mathbf{k}, P)$ to satisfy φ for input $\mathbf{i}$. P' satisfies φ if it does so for all inputs $\mathbf{i}$. Hence, $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k}) = \mathsf{true}$ is also a necessary condition for P' to satisfy φ. Since $\mathsf{OtfRep}(\mathbf{k})$ is always a conjunction of several $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$, it also contains only necessary conditions for P' to satisfy φ. RepairOtf only returns "fail" if $\mathsf{OtfRep}(\mathbf{k})$ is unsatisfiable, so "fail" is only returned if $P' = \mathsf{apply}(\mathbf{k}, P)$ cannot satisfy φ for *any* $\mathbf{k}$, i.e., no repair with the given templates exists.

Statement (b) follows from assumptions about $\textsc{Verify}(P', \varphi)$: it returns $\mathsf{OK} = \mathsf{true}$ iff P' satisfies φ.

Statement (c) also follows from Lemma 39, but read in the other direction: $\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k})$ is true iff $P' = \mathsf{apply}(\mathbf{k}, P)$ satisfies φ for input $\mathbf{i}$. Each candidate program P' either violates φ for some counterexample input $\mathbf{i}$, or RepairOtf terminates. If $P' = \mathsf{apply}(\mathbf{k}, P)$ violates φ for input $\mathbf{i}$, we have that

Algorithm 4.3 REPAIROTF: An algorithm for template-based repair with on-the-fly program analysis.

```
 1: procedure REPAIROTF$\big(P, \varphi, \Delta,$ templates $T_c(\overline{k}_c, \overline{\mathrm{v}}_c)$ for all $c \in \Delta\big)$, returns: A repair $P'$ or “fail”
 2:     $\overline{k} := \bigcup_{c \in \Delta} \overline{k}_c$,    $\overline{\mathrm{k}}$ are fresh program variables corresponding to $\overline{k}$
 3:     $\hat{P}' := \mathsf{apply}(\overline{\mathrm{k}}, P)$
 4:     $\mathsf{OtfRep}(\overline{k}) := \mathsf{true}$
 5:     while true do
 6:         $(\mathsf{sat}, \mathbf{k}) := \textsc{SmtSatModel}\big(\mathsf{OtfRep}(\overline{k})\big)$
 7:         if $\mathsf{sat} = \mathsf{false}$ then
 8:             return “fail”
 9:         end if
10:         $P' := \mathsf{apply}(\mathbf{k}, P)$
11:         $(\mathsf{OK}, \mathbf{i}) := \textsc{Verify}(P', \varphi)$
12:         if OK then
13:             return $P'$
14:         end if
15:         $\mathsf{OtfRep}_{\mathbf{i}}(\overline{k}) := \textsc{Analyze}(\hat{P}', \mathbf{i})$
16:         $\mathsf{OtfRep}(\overline{k}) := \mathsf{OtfRep}(\overline{k}) \wedge \mathsf{OtfRep}_{\mathbf{i}}(\overline{k})$
17:     end while
18: end procedure
```

$\mathsf{OtfRep}_{\mathbf{i}}(\mathbf{k}) = \mathsf{false}$ in Line 15 of REPAIROTF. Hence, after the update of $\mathsf{OtfRep}(\mathbf{k})$ in Line 16, we have that $\mathsf{OtfRep}(\mathbf{k}) = \mathsf{false}$. Since $\mathsf{OtfRep}(\overline{k})$ can only get stricter, $\mathsf{OtfRep}(\mathbf{k})$ will furthermore remain false throughout the entire algorithm. Hence, the same $\mathbf{k}$ cannot be encountered by Line 6 again. □

The statements (a) and (b) from Theorem 40 express that REPAIROTF returns the expected information if it terminates. Statement (c) expresses that REPAIROTF also makes some progress in each iteration. In particular, if $\mathbb{D}$ is finite, then there can only be finitely many assignments $\mathbf{k} \in \mathbb{D}^{|\overline{k}|}$ to the template parameters $\overline{k}$ (because CMP is finite and every template T_c with $c \in \Delta \subseteq \mathsf{CMP}$ can only have finitely many parameters $\overline{k}_c$). Statement (c) implies that REPAIROTF terminates in this case.

4.7.2 Discussion

This section discusses benefits and drawbacks of our repair method with on-the-fly program analysis.

More focused program analysis. The main advantage of the repair method proposed in this section is that program analysis is very focused towards the information needed for computing repair candidates. Complete program analysis is infeasible for complex programs. The basic solution from Section 4.5 can address this issue by setting a limit on the number and length of the execution paths to consider while computing the diagnostic information $\mathcal{D}$. However, since there is no guidance on what to analyze and what to leave out, this limit can render the probability of obtaining the information relevant for finding a (correct) repair rather low. In contrast, our improved repair method analyzes the program only for the counterexamples that are relevant for the repair finding process. There is also a bound on the number and length of the execution paths. However, since these limits apply locally for each invocation, our new approach learns at least something about the behavior under each counterexample.

Simpler program analysis. Compared to the basic solution from Section 4.5, our improved method renders program analysis with symbolic or concolic execution simpler because the inputs are always fixed to one counterexample at a time. This can drastically reduce the number of feasible execution paths. When using concolic execution, it can also simplify the analysis per concolic execution run. We can start to track the symbolic values of the program variables only after a repair template with unknown parameters has been executed for the first time. In particular, when using a reference implementation as a specification, then this is often done by executing both the reference implementation and the program under analysis using the same inputs, and then comparing the results using assertions. If the inputs are always concrete, this means that the entire reference code needs to be executed with concrete variable values only. In contrast, the basic approach without on-the-fly program analysis needs to track the symbolic values right from the beginning because the inputs are not fixed but have a symbolic value.

Flexibility in the specification. The basic solution from Section 4.5 has been designed for assertions in the code as a specification. An extension to test cases is possible, as illustrated in Section 4.6, but defeats some of the strength of the approach. Writing assertions that accurately reflect the desired program behavior and do not only check for basic properties can be difficult, though. Test cases (possibly together with some assertions) are often more natural. Our approach with on-the-fly program analysis supports test cases natively. This flexibility is also important for keeping the user in the loop. Writing additional test cases if only incorrect repairs are produced is often simpler than coming up with better assertions. Our improved approach can even be used in an interactive mode, where no specification is available initially: Input-output examples are provided by the user in order to exclude repair candidates as they are presented.

Scalability. Our improved repair method also addresses the scalability issue, which is common for all formal fault correction approaches, from several sides. Doing program analysis for (typically only a few) concrete counterexamples has already been mentioned. The flexibility in the technique for verifying repair candidates is another factor. Highly optimized model checkers that apply abstraction, specialized reasoning engines or other optimizations may do a better job in verifying repair candidates efficiently than our basic approach of building one monolithic correctness formula and solving it using an SMT solver. Moreover, where formal approaches like model checking or symbolic execution fail, test case execution or manual reviews of candidates by the user can still produce meaningful results.

Drawbacks. A drawback of the separation of concerns is that little information (only the counterexample) is passed from the verification phase to the program analysis phase. Furthermore, if candidate verification is outsourced to an external tool or method, we have no control over the computed counterexamples. In particular, we cannot easily steer the verification towards producing “nasty” counterexamples as we did in our heuristic from Section 4.5.3 in order to speed up the convergence of the repair refinement loop. Another disadvantage is that certain program paths may be feasible under several counterexamples, and may thus be analyzed multiple times using symbolic or concolic execution.

4.8 Parameters and Variations

Now that all steps of our software program repair flow are worked out, we briefly review possible variations and parameters in this section. Our debugging method offers a lot of them. As an advantage, our method can by tailored to a broad range of programs. On the other hand, it may take some attempts to find a good configuration.

Program analysis parameters. In the program analysis step, the parameters include bounds on the analysis depth such as the maximum number of execution paths to analyze or the maximum length of execution paths to consider. Furthermore, different heuristics can be used for determining the order in which program paths are analyzed using symbolic or concolic execution [50]: this ranges from a simple breadth first or depth first search up to advanced heuristics guided by the control flow graph of the program [50]. Another important parameter is the theory used for SMT solving (see Section 2.2.4), which defines the domains $\mathbb{D}$, $\mathbb{T}$ and $\mathbb{F}$. It determines which program language constructs can be handled exactly and which ones have to be approximated. Linear integer arithmetic allows for rather efficient reasoning but cannot (directly) deal with bitwise operations, variable overflows or type casts. Bitvector arithmetic can model such aspects precisely but can be significantly more expensive to reason about.

Alternative correctness conditions. Section 4.2.3.2 suggests to compute the correctness condition

$$\mathsf{Correct}(\bar{i},\bar{r}) = \bigwedge_{n \in \mathsf{FAIL}} \neg \mathsf{PC}_n(\bar{i},\bar{r}). \tag{4.8}$$

An alternative definition would be

$$\mathsf{Correct}'(\bar{i},\bar{r}) = \bigvee_{n \in \mathsf{PASS}} \mathsf{PC}_n(\bar{i},\bar{r}). \tag{4.9}$$

Recall that FAIL is a set of symbolic execution tree nodes. Each node $n \in \mathsf{FAIL}$ represents an analyzed execution paths that ended in a specification violation. The formula PC_n is the path condition associated with node n, i.e., the condition under which the respective execution path is taken. PASS, on the other hand, encompasses all analyzed execution paths that ended in a successful program termination. Finally, OPEN contains the remaining execution paths that were not analyzed completely due to some limit on the analysis depths. If $\mathsf{OPEN} = \emptyset$, i.e., all execution paths have been analyzed, we have that $\mathsf{Correct}$ and $\mathsf{Correct}'$ are equivalent.[10] If certain execution paths were *not* analyzed, there is a difference, though. Equation 4.8 considers a program execution correct if none of the known specification violations is encountered. That is, all execution paths that have not been analyzed are implicitly assumed to satisfy the specification. Consequently, using Equation 4.8 can result in false diagnoses and repairs. We will thus refer to using Equation 4.8 as *non-conservative mode*. On the other hand, Equation 4.9 only considers a program execution correct if it is known to terminate normally. Here, all execution paths that have not been analyzed are implicitly assumed to violate the specification. Hence, using Equation 4.9 can result in missed diagnoses and repairs, and will thus be denoted as *conservative mode*.

Alternative program analysis concepts. Our choice of symbolic or concolic execution for program analysis is mainly motivated by two factors: (1) the multitude of possible approximations and parameters to trade performance for precision, and (2) the success of these techniques in other domains such as test data generation [100]. In principle, the program analysis step to compute the diagnostic information $\mathcal{D}$ can also be realized in completely different ways, though. One alternative option is to unroll all loops for a fixed number of times, transform the program into Static Single Assignment (SSA) form (variable are renamed such that each variable is assigned only once), and apply techniques from bounded model checking [61] in order to obtain the correctness formulas.

Fault localization parameters. The most important parameter to configure the precision of our fault localization approach from Section 4.3 is the number $|J|$ of concrete counterexamples to consider. In the model-based diagnosis algorithm, we can also set a limit on the maximum number and the maximum size of diagnoses to compute. This is important in order not to flood the user (and the repair engine) with diagnoses (of high cardinality).

Fault correction parameters. The parameters for our basic fault correction method from Section 4.5 include the templates to use, a maximum number of counterexample-guided repair refinements before giving up, and a maximum number of repairs to compute (in total and/or per diagnosis). The repair method with on-the-fly program analysis (Section 4.7) additionally can be configured with all parameters that have already been discussed for program analysis. This also includes the option to compute the more conservative correctness formula

$$\mathsf{OtfRep}'_{\mathbf{i}}(\bar{k}) = \bigvee_{n \in \mathsf{PASS}} \mathsf{PC}_n(\bar{k}). \tag{4.10}$$

instead of $\mathsf{OtfRep}_{\mathbf{i}}(\bar{k})$ as defined in Equation 4.7.

Additional program analysis passes. As already argued in Section 4.7, performing the program analysis step only once in the beginning has the disadvantage that the resulting diagnostic information $\mathcal{D}$ may be relatively imprecise (due to bounds on the program analysis depth) for a particular diagnosis Δ to repair. Our repair approach with on-the-fly program analysis (Section 4.7) addresses this issue by analyzing the program anew, but only for the specific counterexamples that are encountered during the repair

[10] The intuitive reason is that the preprocessed program $\hat{P}$ can only either violate or satisfy its specification φ for given values $\mathbf{i}, \mathbf{r}$ of the symbols $\bar{i}, \bar{r}$. The more formal reason is that all path conditions are disjoint (i.e., $\mathsf{PC}_{n_1}(\bar{i},\bar{r}) \wedge \mathsf{PC}_{n_2}(\bar{i},\bar{r})$ is unsatisfiable if $n_1 \neq n_2$), FAIL and PASS are disjoint, and $\bigvee_{n \in \mathsf{PASS} \cup \mathsf{FAIL}} \mathsf{PC}_n(\bar{i},\bar{r})$ is true if $\mathsf{OPEN} = \emptyset$.

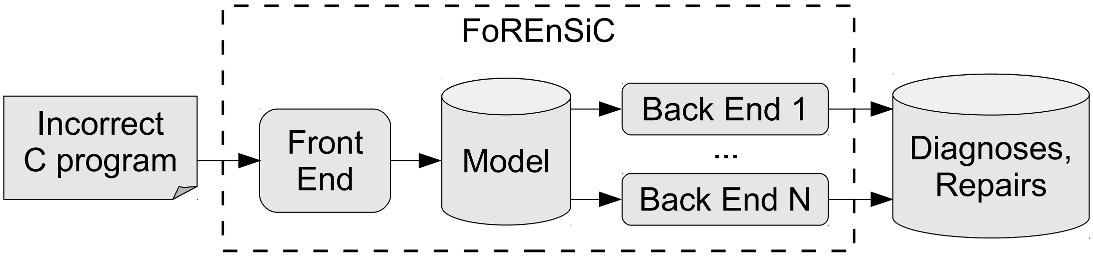


Figure 4.3: Architecture of FoREnSiC. The main input is an incorrect C program. The front end creates an internal model of the program and the back ends use this model to compute diagnoses and repairs with different techniques.

process. A more lightweight approach to tackle the problem is to perform only one additional program analysis pass per diagnosis to repair. In this pass, the components that are not repaired get fixed to their original implementation. This reduces the number of feasible execution paths and results in diagnostic information $\mathcal{D}_\Delta$ that is tailored towards the specific diagnosis Δ to repair. The disadvantage compared to our approach of one-the-fly program analysis is that the diagnostic information $\mathcal{D}_\Delta$ is not tailored towards the counterexamples that will be encountered during the repair process. The advantage, however, is that program analysis needs to be done only once per diagnosis and not once per counterexample in the repair process.

4.9 Experimental Results

The previous sections in this chapter explained *how* our approach for controller synthesis in the application of software repair works. The aim of this is section is to evaluate *to which extent* it works in practice. To this end, we will first present our proof-of-concept implementations in the tools FoREnSiC [30] and Frama-C [69]. Section 4.9.2 will then discuss some examples to demonstrate that our approach is able to produce helpful diagnoses and repairs in reasonable time. Section 4.9.3 contains a performance evaluation of our approach in different configurations. Section 4.9.4 finally draws conclusions.

4.9.1 Implementation

The debugging approach outlined in Figure 4.1 has been implemented as a proof-of-concept in FoREnSiC [30], which is a debugging environment for simple C programs. The fault localization variant using deductive verification (Section 4.4) has been prototyped as an extension to the WP plug-in of Frama-C [69]. This extension will be called WPLoc and is presented in Section 4.9.1.2.

4.9.1.1 FoREnSiC

We developed FoREnSiC [30] in collaboration with the University of Bremen and Tallinn University of Technology. The name FoREnSiC stands for "**Fo**rmal **R**epair **En**vironment for **Si**mple **C**". Just like Demiurge, FoREnSiC is also freely available under the GNU Lesser General Public License (version 2.1). It can be downloaded from

`http://www.informatik.uni-bremen.de/agra/eng/forensic.php`.

All experiments presented in this thesis have been performed using version `1.0.1`. The downloadable archive contains all scripts to reproduce the experiments, as well as spreadsheets with more detailed data. FoREnSiC features a modular architecture with different back ends implementing different debugging engines. In the following, we will briefly describe the general architecture and then focus on the symbolic back end, which implements the debugging approach of Figure 4.1.

Architecture. Figure 4.3 illustrates the basic architecture of FoREnSiC. The main input is a (potentially) incorrect C program. A front end based on the gcc compiler parses this program and constructs an internal model. This internal model is essentially a flow graph where nodes correspond to statements of the program and edges model the control flow. Each node also stores an Abstract Syntax Tree (AST) of the respective statement as well as source code location information in terms of line and column numbers. This is important for communicating results back to the user. Different back ends finally implement different fault localization and correction algorithms based on this model. The back ends may read additional inputs such as specifications in the form of test cases or a reference model. They can access a library of utility functions as well as interfaces to reasoning engines such as SMT solvers. This architecture and infrastructure makes FoREnSiC easily extendable with new back ends and features.

The following paragraphs describe the symbolic back end, which implements the debugging approach of Figure 4.1. The variant using test cases as a specification (Section 4.6) has not yet been implemented. Test cases are only supported for repair with on-the-fly program analysis.

Concolic execution. By default, program analysis is done via concolic execution. We implemented our concolic execution engine as an extension to CREST [50] and thus inherit all its different search strategies and heuristics. CREST itself supports only Linear Integer Arithmetic (LIA) as theory for SMT solving. Program language constructs that cannot be modeled in LIA (e.g., bitwise operations) are abstracted by taking the fixed values from the concrete part of the execution. CREST uses the SMT solver Yices [77]. Our extension supports also the SMT solver Z3 [73] as well as the Bitvector Arithmetic (BV) theory for SMT solving. The program under analysis can also contain floating point variables, arrays, pointers, and compound data types like `structs` and `unions`. However, these constructs are only handled in an approximative way. Arrays are essentially treated like sets of variables. That is, the values of array elements are tracked symbolically, but array indices are always abstracted with their concrete value. This has the effect that our implementation cannot diagnose or repair bugs in array index computations. The same holds for pointers: the value of the pointer target is tracked symbolically but arithmetic applied to the pointer itself is always abstracted with the resulting concrete pointer value. Consequently, bugs in pointer arithmetic cannot be diagnosed and repaired either. Floating point variables in the program are completely ignored in the symbolic analysis.

Preprocessing for concolic execution. Section 4.2.1 proposed to take all expressions in the code as potentially faulty components. Our implementation does not follow this idea strictly. It only takes all conditions in the program as well as the right-hand side (RHS) of all assignments that have an integer result as potentially faulty components. However, the front end[11] simplifies the program by assigning non-trivial expressions to auxiliary variables before they are used. Consequently, our implementation also considers them as potentially faulty components. If the user encapsulates code blocks with `<ASSUME_CORRECT>` and `</ASSUME_CORRECT>` tags, this code will always be assumed to be correct, i.e., it will not be instrumented with calls to `cmpa` or `cmpc`. This is useful, for instance, when using reference implementations as a specification.

Symbolic execution. A symbolic execution engine for computing the diagnostic information $\mathcal{D}$ has been implemented by Philipp Pani in the frames of his Bachelor's Thesis [170]. It can be used as an alternative to the concolic engine. Just like the concolic engine, it supports the SMT solvers Yices [77] and Z3 [73] with LIA or BV. Operations that cannot be modeled in the chosen theory (e.g., a division when using LIA) are abstracted using a fresh input symbol that can take any value. In contrast to the concolic engine, the symbolic execution engine does not support floating point variables, arrays, pointers, and compound data types like `structs` and `unions`.

Fault localization. We implemented the fault localization method from Section 4.3.2 in several variants. All variants use the hitting set tree algorithm by Reiter [182] for deriving diagnoses from conflicts but different methods to compute minimal conflicts. A basic variant computes minimal conflicts

[11] When using concolic execution, we actually bypass the gcc-based front end shown in Figure 4.3 and use CIL [167] instead. The reason is that CREST [50] also uses CIL in order to instrument the program for concolic execution. The gcc-based front end performs similar simplifications, though.

by dropping components one after the other as long as the reduced set is still a conflict. This is realized using one SMT solver call per attempt to drop a component. A more advanced approach implements the procedure GETMINCONFWO from Algorithm 4.1 utilizing unsatisfiable cores computed by the SMT solver but without incremental SMT solving. A third variant also uses incremental SMT solving (even across multiple calls to GETMINCONFWO).

Templates for fault correction. Let $c \in \Delta$ be an arithmetic component (i.e., $\mathrm{TypOf}(c) = \mathbb{D}$) to repair and let $\overline{\mathtt{v}}_c = \mathrm{Vars}(c) = (\mathtt{v1}, \mathtt{v2}, \ldots, \mathtt{vn})$ be the variables in scope when component c is executed. When LIA is chosen as a theory for SMT solving, we use the linear template

$$k_0 + k_1 \cdot \mathtt{v1} + k_2 \cdot \mathtt{v2} + \ldots + k_n \cdot \mathtt{vn}$$

to find a new implementation for component c. When BV is chosen as the SMT theory, the following sequence of templates is tried:

Name	Template
linear	$k_0 + k_1 \cdot \mathtt{v1} + k_2 \cdot \mathtt{v2} + \ldots + k_n \cdot \mathtt{vn}$
dnf	$k_0 \ \vert\ (k_1 \ \mathtt{\&}\ \mathtt{v1}) \ \vert\ (k_2 \ \mathtt{\&}\ \mathtt{v2}) \ \vert\ \ldots \ \vert\ (k_n \ \mathtt{\&}\ \mathtt{vn})$
cnf	$k_0 \ \mathtt{\&}\ (k_1 \ \vert\ \mathtt{v1}) \ \mathtt{\&}\ (k_2 \ \vert\ \mathtt{v2}) \ \mathtt{\&}\ \ldots \ \mathtt{\&}\ (k_n \ \vert\ \mathtt{vn})$
shift-dnf	$k_0 \ \vert\ \Big(k_1 \ \mathtt{\&}\ \big((\mathtt{v1}\mathtt{<<}k_2)\mathtt{>>}k_3\big)\Big) \ \vert\ \ldots \ \vert\ \Big(k_{3n-2} \ \mathtt{\&}\ \big((\mathtt{vn}\mathtt{<<}k_{3n-1})\mathtt{>>}k_{3n}\big)\Big)$
shift-cnf	$k_0 \ \mathtt{\&}\ \Big(k_1 \ \vert\ \big((\mathtt{v1}\mathtt{<<}k_2)\mathtt{>>}k_3\big)\Big) \ \mathtt{\&}\ \ldots \ \mathtt{\&}\ \Big(k_{3n-2} \ \vert\ \big((\mathtt{vn}\mathtt{<<}k_{3n-1})\mathtt{>>}k_{3n}\big)\Big)$

Here, "&" denotes a bit-wise conjunction, "|" denotes a bit-wise disjunction, and "<<" and ">>" are bitshift operations. Templates for conditional components (with $\mathrm{TypOf}(c) = \mathbb{B}$) are always of the form `T OP 0`, where `T` is a template for an arithmetic component and $\mathtt{OP} \in \{>, <, >=, <=, ==, !=\}$ is a comparison operator. A template parameter encodes which of the comparison operators is taken. Defining additional or alternative templates is easily possible.

Basic fault correction. Our implementation of the basic fault correction approach from Section 4.5 is highly configurable. Features that can be enabled or disabled include (1) our heuristic for computing simple repair candidates and nasty counterexamples (Section 4.5.3), (2) incremental SMT solving in the CEGIS loop, and (3) an additional program analysis pass (see Section 4.8) per diagnosis to obtain more accurate diagnostic information. In addition to bounds on the maximum number of repair refinements, the user can also set a timeout to all SMT solver calls in order to ensure termination.[12] We also implemented a few optimizations to the CEGIS approach. One optimization reduces the set of counterexamples (D in Figure 2.5) from time to time: counterexamples are removed as long as D still contains at least one counterexample to disprove every candidate that has been encountered so far. This makes the formula for candidate computation smaller. However, in experiments we have observed that this optimization can increase the number of refinements that are necessary to find a repair. The reason is that even if a counterexample is not necessary to disprove the already encountered candidates, it may still be useful for ruling out not yet encountered candidates. This heuristic is therefore disabled by default. Another optimization adds random input vectors to the counterexample database D from time to time. While this can decrease the number of repair refinement iterations, it can also result in larger formulas for candidate computation. This optimization is therefore disabled by default as well.

Repair with on-the-fly program analysis. Two methods for verifying repair candidates are currently supported: test case execution and concolic execution. On-the-fly program analysis is realized using concolic execution. Incremental SMT solving as well as our heuristic from Section 4.5.3 can again be enabled or disabled.

[12] A timeout during repair candidate computation is handled as if no candidate exists, i.e., the procedure aborts. In case of a timeout during candidate verification, the tool gives a warning but considers the candidate to be correct.

Other variants. We have also implemented a few alternatives to the CEGIS approach for template-based repair computation. A brute-force method takes as input a set L of likely values (such as 0, 1, -1, etc.) for template parameters. First, it computes all possible vectors of template parameter values $\mathbf{k}$, with individual parameter values taken from L, up to a given bound. Heuristics sort these vectors for maximum success likelihood. Then, for one vector $\mathbf{k}$ after the other, the brute-force method checks if this vector constitutes a correct repair. Another method passes the quantified formula $\exists \overline{k} : \forall \overline{i} : \mathsf{Rep}(\overline{i}, \overline{k})$ to the SMT solver Z3 and lets the solver deal with the quantifier alternation directly. However, compared to our CEGIS approach, none of these alternatives performed well in preliminary experiments.

4.9.1.2 Fault Localization Using Deductive Verification

We implemented the alternative fault localization approach from Section 4.4 as a proof of concept in the WP plug-in of the widely used software analysis tool suite Frama-C. This implementation will be called WPLoc and can be downloaded from

http://www.iaik.tugraz.at/content/research/design_verification/others/.

At the moment, only the computation of single-fault diagnoses is supported. An extension to computing diagnoses with higher cardinality is conceptually simple, though. Several challenges had to be resolved in our implementation. These challenges are discussed in the following paragraphs. Finally, we discuss reasons for imperfect diagnostic resolution in our implementation.

Preprocessing. Recall from Section 4.4 that our approach replaces expressions in the code with fresh repair variables in a preprocessing step. We use fresh global program variables to model these repair variables. Frama-C normalizes the source code while parsing it into an Abstract Syntax Tree (AST). For instance, it rewrites all loops into simple while-loops, it decomposes complicated statements using auxiliary variables, etc. Our preprocessing works on this normalized AST. This makes it robust when handling complicated constructions. The disadvantage is that our approach may report fault locations that are not present in the original program but were only introduced by the normalization. However, we do not consider this a severe usability issue because the line number in the original code is available. Also, Frama-C presents the normalized source code and how it links to the original code in its GUI.

Computation of $\mathsf{Corr}(\overline{i}, \overline{r})$**.** Unfortunately, we cannot use the WP plug-in of Frama-C as a black-box to compute the correctness formula $\mathsf{Corr}(\overline{i}, \overline{r})$ after preprocessing. WP performs simplifications that may rewrite or eliminate our newly introduced repair variables $\overline{r}$. We solve this issue by extending Frama-C's memory model such that the repair variables $\overline{r}$ are not touched by formula simplifications.

Quantification. Once we have $\mathsf{Corr}(\overline{i}, \overline{r})$ available, our approach adds the quantifier prefix $\forall \overline{i} : \exists \overline{r}$. Unfortunately, Corr may also contain auxiliary variables $\overline{t}$ that express values of variables at specific program points. Intuitively, the values of the symbols $\overline{r}$ should not depend on variables that are assigned later in the program. This would violate the causality and lead to false-positives. We therefore separate the variables of Corr to construct the formula $\forall \overline{i} : \exists \overline{r} : \forall \overline{t} : \mathsf{Corr}(\overline{i}, \overline{r}, \overline{t})$. This is done by computing the input variables (parameters and global variables) of the function under analysis and linking them to the corresponding variables in the formula. Variables in the formula that cannot be linked to an input of the function are put into inner quantification $\forall \overline{t}$.

Axiomatization. WP uses axiomatized functions and predicates in Corr. For instance, instead of $a < b$, WP writes $\mathrm{zlt}(a, b)$ in the constructed formulas, where the predicate $\mathrm{zlt} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$ is axiomatized as $\forall x, y : \big(zlt(x, y) \rightarrow x < y\big) \wedge \big(\neg zlt(x, y) \rightarrow x \geq y\big)$. In our experiments we observed cases where the automatic theorem prover (we used Alt-Ergo[13]) could not solve formulas when using the axiomatization, but had no difficulty solving the same formula when the axiomatized predicates and functions are replaced by the corresponding native operators. Hence, we modified the interface to the theorem prover such that formulas do not contain axiomatized functions and predicates, where possible.

[13] http://alt-ergo.lri.fr (last visit on 2015-08-01).

Diagnostic resolution. Our implementation is neither sound (it may report spurious fault locations) nor complete (it may miss potential fault locations). The main reasons are as follows.

- The theorem prover may return “Unknown” if it could neither prove nor disprove the validity of the formula. We treat such verdicts as if the program was incorrect (see Section 4.4), which results in incompleteness.
- Instead of one monolithic formula Corr, WP may compute multiple formulas that can be checked independently. In fault localization, we also check repairability for each formula in isolation. This is weaker than checking the conjunction and can thus result in reporting spurious fault locations but increases efficiency.
- The specification may be incomplete, i.e., not strict enough. This can result in reporting spurious fault locations.
- The bug in the program may not match our fault model. For instance, code may be missing or the control flow may be incorrect. This results in missed fault locations.

Despite these potential imprecisions, our implementation usually produces meaningful results. This is illustrated in the next section.

4.9.2 Examples

This section presents a few code examples and discusses output that is produced by our tools on a qualitative level. The aim is to demonstrate that our approach is able to produce helpful diagnoses and repairs in reasonable time. Section 4.9.3 will then discuss the performance of our approach in various configurations.

4.9.2.1 Running Example

For the program from Example 22, FoREnSiC produces the diagnoses $\Delta_1 = \{c_3\}$ and $\Delta_2 = \{c_1, c_2\}$ that were already discussed in Example 30. That is, the tool suspects either the expression “a + 2” in Line 4 to be faulty, or the two expressions “a” and “b > a” in Line 2 and 3 to be simultaneously faulty. With a limit of three corrections per diagnosis and LIA as SMT theory, FoREnSiC produces the following repair suggestions.

```
1 int max(int a, int b){
2  int r = a;
3  if(b > a)
4   r = b; //a+2
5  assert(r>=a && r>=b);
6  return r;
7 }
```

```
int max(int a, int b){
 int r = a;
 if(b > a)
  r = b + 1000; //a+2
 assert(r>=a && r>=b);
 return r;
}
```

```
int max(int a, int b){
 int r = a;
 if(b > a)
  r = b + 1; //a+2
 assert(r>=a && r>=b);
 return r;
}
```

```
1 int max(int a, int b){
2  int r = b; //a
3  if(a > b)  //b>a
4   r = a + 2;
5  assert(r>=a && r>=b);
6  return r;
7 }
```

```
int max(int a, int b){
 int r = b + 1; //a
 if(a > b)       //b>a
  r = a + 2;
 assert(r>=a && r>=b);
 return r;
}
```

```
int max(int a, int b){
 int r = b + 1000; //a
 if(b < a)          //b>a
  r = a + 2;
 assert(r>=a && r>=b);
 return r;
}
```

The entire computation takes only three seconds. Some of the suggested repairs may be unsatisfactory for a function that is supposed to compute the maximum of two integers. The reason is the incomplete specification. If we refine this specification to also require that r must be equal to either a or b, we get repairs that suggest to replace a + 2 by b, by a + b - r, by -a + b + r, by 3*a + b - 3*r, by `4*a + b - 4*r`, etc. Since `r=a` holds at Line 4, all these suggestions are equivalent to the first one and, hence, obviously correct.

Our extension WPLoc of Frama-C's WP plug-in computes only single-fault diagnoses. Hence, it does not report $\Delta_2 = \{c_1, c_2\}$ but only $\Delta_1 = \{c_3\}$ as well as the expression "`r`" in Line 6 (because the specification refers to the returned value and not to the content of variable `r`; see Example 33). This takes only two seconds.

4.9.2.2 Traffic Collision Avoidance System (TCAS)

Our next example is the Traffic Collision Avoidance System (TCAS) for aircrafts from the Siemens benchmark suite.[14] It has first been used by Hutchins et al. [114] and can be seen as a standard benchmark for automatic debugging. The TCAS program has 137 lines of code (not counting comments and blank lines) spread over 8 functions, 12 integer inputs and one output. It comes in 41 faulty versions as well as a reference implementation and 1608 test cases. When running FoREnSiC, we use an assertion that compares the computed result with that of the reference implementation as a specification. For WPLoc, we wrote pre- and postconditions for all functions in the code. The TCAS program does not contain any loops, so writing loop invariants was not necessary for WPLoc.

Version 2. The faulty version number 2 contains the following function:

```
75 int Inhibit_Biased_Climb () {
77   return (Climb_Inhibit ? Up_Separation + MINSEP : Up_Separation);
78 }
```

Here, `MINSEP` is a constant (implemented as a preprocessor macro) with value 300. The correct reference implementation uses the constant `NOZCROSS` (value 100) instead of `MINSEP`. WPLoc reports two potential fault locations, where one of them is the expression `Up_Separation + MINSEP` in Line 77. This takes only 8 seconds. Using LIA, FoREnSiC produces 11 diagnoses, whereof 3 are single-fault diagnoses[15] and the remaining 8 have a cardinality of 2. With a bound of 3 on the maximum number of repairs to compute, FoREnSiC suggests to replace `Up_Separation + MINSEP` in Line 77 by

- `Up_Separation + 100`,
- `-Down_Separation + 2*Up_Separation + 199`, or
- `-6*Down_Separation + 7*Up_Separation + 700`.

The first repair directly corresponds to the code in the reference implementation. The latter two repairs are correct (which is also confirmed by the model checker CBMC [61]) as well because the function `Inhibit_Biased_Climb()` is only called in comparisons of the form `Inhibit_Biased_Climb() > Down_Separation`. With the second repair, this comparison becomes

$$\begin{aligned} & \texttt{-Down\_Separation + 2*Up\_Separation + 199 > Down\_Separation} \\ = \; & \texttt{2*Up\_Separation + 199 > 2*Down\_Separation} \\ = \; & \texttt{Up\_Separation + 100 > Down\_Separation}. \end{aligned}$$

The last equality only holds because all variables are integers and program variable overflows are disregarded when using LIA. The same reasoning applies to the third repair. FoREnSiC takes around 80 seconds to find these solutions.

[14] `http://sir.unl.edu/portal/bios/tcas.php#siemens` (last visit on 2015-08-01).

[15] Recall that WPLoc only performs fault localization for the functions that violate their local specification. FoREnSiC takes a global view of the program and can also identify components outside of `Inhibit_Biased_Climb` as potentially faulty. This is one reason for the difference in the number of reported single-fault diagnoses.

Table 4.1: Example repair process for TCAS version 28 using FoREnSiC. We compute three repairs with the linear repair template. The first column gives the iteration number, the next two columns the candidate replacements, and the last column indicates whether the candidate was found to be correct or not. Even though the search space for expressions is huge, correct solutions are found quickly.

Iteration	Candidate replacement for	Candidate replacement for	Correct
	Up_Separation + NOZCROSS	Up_Separation	
1	0	0	✗
2	0	728	✗
3	-1000	Up_Separation + 1000	✗
4	-1000	Up_Separation + 100	✗
5	-32768	High_Confidence + 1000	✗
6	-32768	Up_Separation + 100	✗
7	-Other_Tracked_Alt + 1	Other_Tracked_Alt - 260	✗
8	-Other_Tracked_Alt + 1	-Other_RAC - 615	✗
9	Down_Separation - 1000	-Climb_Inhibit + 1000	✗
10	Down_Separation - 1000	Up_Separation - 1000	✗
11	Up_Separation - 1000	Own_Tracked_Alt - 262	✗
12	Down_Separation - 1000	Other_Tracked_Alt - 261	✗
13	Down_Separation - 1000	Up_Separation - 999	✗
14	Up_Separation	Up_Separation - 998	✗
15	Up_Separation - 1000	Up_Separation + 1	✗
16	Up_Separation - 1000	Up_Separation + 100	✗
17	-Other_Tracked_Alt + 1	Up_Separation + 100	✗
18	Up_Separation	Up_Separation + 100	✓
19	Up_Separation	Up_Separation + 2	✗
20	Up_Separation	Up_Separation + 99	✗
21	Down_Separation - Other_RAC - 229	Up_Separation + 100	✗
22	Down_Separation - Own_Tracked_Alt - 1	Up_Separation + 100	✗
23	-Own_Tracked_Alt + Up_Separation - 1000	Up_Separation + 100	✗
24	-Other_RAC + Up_Separation - 229	Up_Separation + 100	✗
25	ALV + Up_Separation - 1	Up_Separation + 100	✗
26	Up_Separation	-ALV + Up_Separation + 101	✗
27	-ALV + Up_Separation + 1	Up_Separation + 100	✗
28	Climb_Inhibit + Up_Separation	Up_Separation + 100	✓
29	Up_Separation	ALV + Up_Separation + 99	✗
30	-Climb_Inhibit + Up_Separation	Up_Separation + 100	✓

Version 28. In this version of the TCAS program, Inhibit_Biased_Climb is implemented as:

```
int Inhibit_Biased_Climb () {
  return (Climb_Inhibit == 0 ? Up_Separation + NOZCROSS : Up_Separation);
}
```

In the reference implementation, the condition Climb_Inhibit == 0 of the ternary if is negated. WPLoc takes 10 seconds to report two diagnoses, one of which is this condition. FoREnSiC reports 20 diagnoses and computes the expected fix within 30 seconds when run with appropriate parameters. However, with suboptimal parameters (running the diagnosis engine in the conservative mode using Equation 4.9 with too few execution paths analyzed by concolic execution), FoREnSiC does not find out that Climb_Inhibit == 0 may be wrong, but only that the if-part Up_Separation + NOZCROSS and the else-part Up_Separation of the ternary if may be wrong simultaneously. We take this as an opportunity to illustrate the strength of our repair engine by showing that it finds a fix anyway.

Table 4.1 shows the repair candidates that are computed by the CEGIS loop when asking FoREnSiC to synthesize three repairs using the linear repair template. Since the TCAS program has 12 global integer variables, we have 26 template parameters for repairing the two expressions simultaneously. But even though the parameter space is huge, the first solution is found already after 18 iterations. It corresponds to swapping the `if`-part and the `else`-part of the ternary `if` in Line 77 of the TCAS program. Since FoREnSiC is requested to compute two more repairs, it still continues after Iteration 18. The additional repairs found in Iteration 28 and 30 are also correct because the `if`-part is only evaluated if `Climb_Inhibit` is `0`. The repair process takes only 80 seconds, the entire execution time is 130 seconds. Our heuristic from Section 4.5.3 is crucial for CEGIS to converge to a solution so quickly. If this heuristic is disabled, the tool just investigates utterly complex candidate expressions (mostly containing all 12 global program variables in both expressions) until a solver timeout is hit.

4.9.2.3 Greatest Common Divisor Example

The following source code listing shows an optimized function (`gcd`) for computing the greatest common divisor of two integer numbers on the right-hand side. This function contains a bug which is not easy to see and even more difficult to fix: Line 41 should read `u - v` instead of `u >> 1`. The standard Euclidean algorithm (function `gcdR`) is used as a reference implementation on the left-hand side.

```
 1  #include <assert.h>
 2  #include <forensic.h>
 3  #define UI unsigned int
 4
 5  //<ASSUME_CORRECT>
 6  UI gcd(UI u, UI v);
 7  UI gcdR(UI a, UI b) {
 8    if(a == 0)
 9      return b;
10    while(b != 0){
11      if(a > b)
12        a = a - b;
13      else
14        b = b - a;
15    }
16    return a;
17  }
18  void main() {
19    UI a, b;
20    FORENSIC_input_UI(a);
21    FORENSIC_input_UI(b);
22    assert(gcdR(a,b) == gcd(a,b));
23  }
24  //</ASSUME_CORRECT>
```

```
25  UI gcd(UI u, UI v) {
26    UI shift = 0;
27    if(u == 0 || v == 0)
28      return u | v;
29    for(;((u|v)&1)==0; ++shift) {
30      u >>= 1;
31      v >>= 1;
32    }
33    while((u & 1) == 0)
34      u >>= 1;
35    do {
36      while((v & 1) == 0)
37        v >>= 1;
38      if(u <= v) {
39        v -= u;
40      } else {
41        UI tmp = u >> 1;
42        u = v;
43        v = tmp;
44      }
45      v >>= 1;
46    } while(v != 0);
47    return u << shift;
48  }
```

This example is challenging for FoREnSiC in various respects. First, it mixes arithmetic operations with bitwise operation. We will thus run FoREnSiC with bitvector arithmetic as theory for SMT solving. Second, due to the many loops in the program, the number of possible execution paths is huge, which makes exhaustive program analysis using symbolic or concolic execution infeasible.

Fault localization. The fault localization results are not particularly impressive: with rather low parameters for the program analysis depth, FoREnSiC reports 9 of the 15 components that are identified in `gcd` as potentially faulty. This takes only 4 seconds. Higher parameters for the program analysis depth reduce the number of diagnoses to 6, but this computation takes already 120 seconds.

Fault correction with our basic method. Our basic method for fault correction (Section 4.5) does not work at all for this example. It either finds no repair (when using the conservative mode, i.e., Equation 4.9), incorrect repairs (when using the non-conservative mode, i.e., Equation 4.8), or the solver hits a timeout (when choosing too high analysis depth parameters).

Fault correction with on-the-fly program analysis. Our repair method with on-the-fly program analysis (Section 4.7) and test cases as a specification still performs well. As test inputs, we take all pairs a, b with $0 \leq a, b < 100$. The number 100 was chosen arbitrarily, the correct repair is also found with lower numbers like 15. For program analysis, we do not limit the number of execution paths to analyze, but rather their length. With two or three invocations of our method, we found out that a length of 55 is enough.[16] Using these parameters and Z3 with bitvector arithmetic, our method encounters the sequence of repair candidates "`873`", "`12`", "`u - 2`", and "`u - v`" in 80 seconds. The last one, "`u - v`", was found to be correct.

4.9.2.4 DLX Processor

Our next example is a C program that emulates a DLX processor, which is a RISC processor architecture introduced by Patterson and Hennessy in a textbook on computer architecture [172]. Assertions in the code are used as a specification. Including assertions but excluding blank lines and comments, the processor emulator has 526 lines of code. Instructions are executed in three phases. The FETCH phase reads the next instruction from memory, the DECODE phase decomposes it, and the EXECUTE phase finally executes the instruction. For our debugging experiment with FoREnSiC, we place only one instruction in memory. This instruction (4 byte) is modeled as an input. Bitvector arithmetic is used as theory for SMT solving.

Bug in instruction fetching. The first bug we consider is in a function that reads a 32 bit word from memory. It is called when fetching the next instruction from memory.

```
442 unsigned read32(unsigned addr) {
443   unsigned res = mem[addr];
444   res <<= 8;
445   res |= mem[addr+1];
446   res <<= 8;
447   res |= mem[addr+2];
448   res <<= 8;
449   res |= mem[addr+3];
450   res <<= 8; //BUG: remove.
451   return res;
452 }
```

```
void fetch() {
 ir = read32(pc);
 assert(ir & 0xFF == mem[pc+3]);
 assert(ir & 0xFF00
        == mem[pc+2] << 8);
 assert(ir & 0xFF0000)
        == mem[pc+1] << 16);
 assert(ir & 0xFF000000
        == mem[pc] << 24);
 pc += 4;
}
```

Line 450 contains a copy-paste mistake: this line should be removed. With the assertions in the `fetch()` function, the bug is easy to detect, locate and repair for FoREnSiC. With a limit of 3 repairs per diagnosis, FoREnSiC reports 16 repairs within a total execution time of only 5 seconds. Among them are suggestions to replace the statement in Line 450 by `res = res;` or `res = 4294967295 & res;`, which are both equivalent to removing this statement (because `4294967295` is `0xFFFFFFFF` in hexadecimal representation). Other repairs make the processor halt (and the C program terminate) before `fetch()` is even called. This successfully prevents an assertion violation because the assertion is never executed. Such repairs are clearly undesirable but result from the way we define correctness with respect to assertions. We will leave this issue for future work (see Section 6.3.2).

[16] This number roughly corresponds to the number of executed statements.

Bug in instruction decoding. The next bug we consider is a faulty bitmask in Line 102 of the instruction decoding routine. Assertions again serve as a specification:

```
 94 void decode() {
 95   opcode = ir >> 26;
 96   rega = (ir >> 21) & 0x1F;
 97   regb = (ir >> 16) & 0x1F;
 98   regc = (ir >> 11) & 0x1F;
 99   uimm16 = ir & 0xFFFF;
100   imm16 = uimm16;
101   if((uimm16 & 0x8000) != 0)
102     imm16 |= 0xFFFFF000;   // correct: |= 0xFFFF0000
103   unsigned uimm26 = ir & 0x3FFFFFF;
104   imm26 = uimm26;
105   if((imm26 & 0x02000000) != 0)
106     imm26 |= 0xFC000000;
107   func = ir & 0x7FF;
108 
109   // Some sanity checks:
110   assert((opcode & 0xFFFFFFC0) == 0); // opcode has only 6 bit
111   // ... some more assertions are not printed here ...
112   // imm16 and uimm16 must represent the same 16 bit value
113   assert((uimm16 & 0xFFFF) == (imm16 & 0xFFFF));
114   // a negative number implies that sign extension has been done:
115   assert((uimm16 & 0x8000) == 0 || (imm16 & 0xFFFF0000) == 0xFFFF0000);
116   // a non-negative number implies that the upper bits are zero:
117   assert((uimm16 & 0x8000) != 0 || (imm16 & 0xFFFF0000) == 0);
118 }
```

With a limit of 5 repairs per diagnosis, FoREnSiC takes only 18 seconds to produce 25 repairs. Some repairs again make the processor halt before `decode()` is reached. The repairs for Line 102 suggest a replacement with one of the following statements[17]:

(1) `imm16 = 0xFFFF0000 | ir;`

(2) `imm16 = 0xFFFF8000 | ir;`

(3) `imm16 = 0xFFFF0000 | uimm16;`

(4) `imm16 = 0xFFFF0000 | imm16;`

(5) `imm16 = 0xFFFF8000 | imm16;`

Repair (4) corresponds to the expected solution. Repair (3) is equivalent because `uimm16` is equal to `imm16` at Line 102. Repair (5) is also correct because Line 102 is only executed if bit number 16 is set in `uimm16` and `imm16`. Hence, there is no difference between the bitwise OR with `0xFFFF8000` and `0xFFFF0000`. The first two repairs are correct as well because `imm16` and `ir` share the same lower 16 bit. In summary, even though the function `decode()` is rather short, its code is all but trivial. Detecting, locating and fixing subtle bugs like wrong bitmasks can be very cumbersome for a human developer. FoREnSiC takes only a few seconds to suggest reasonable fixes automatically. More examples of bugs in the DLX processor design can be found in the downloadable FoREnSiC archive.

4.9.3 Performance Evaluation

In this section, we evaluate and compare the performance and the diagnostic resolution of our approach in various configurations. All experiments are performed on a notebook with an Intel Core i5-3320M processor running at 2.6 GHz, 8GB of RAM, and a 64 bit Linux operating system. We use FoREnSiC

[17] Constants have been transformed into hexadecimal representation manually and bitwise ANDs with `0xFFFFFFFF` have been removed manually in order to increase readability.

always with the SMT solver Z3 version 4.4.0 via its API. Our extension WPLoc of Frama-C's WP plug-in uses the theorem prover Alt-Ergo version 0.95.

4.9.3.1 Fault Localization Results

Table 4.2 summarizes performance results for fault localization on the 41 faulty versions of the TCAS benchmark. See Section 4.9.2.2 for a description of this benchmark. Column 1 lists the TCAS version number. Column 2 indicates whether the bug in the respective version matches our fault model of incorrect expressions. This is the case for around 60 % of the versions. The remaining 40 % contain wrong control flow[18], missing code or wrong array sizes or indices. The number of source code modifications compared to the reference implementation is given in Column 3. Most of the introduced bugs are single-faults but some versions contain several simultaneously faulty program parts. The Columns 4 and 5 give the execution time and the number of potential fault locations reported by the tool Bug-Assist [130], which serves as a baseline for comparison. The Columns 6 to 9 give results for FoREnSiC when running fault localization in the non-conservative mode (using Equation 4.8). The maximum number of execution paths to analyze in the program analysis step was limited to 400, the number $|J|$ of input vectors was set to 3, and the diagnosis engine was run with all optimizations (unsatisfiable cores and incremental solving). The program analysis time (not listed in Table 4.2) is around 16 seconds for all versions. The fault localization time is given in Column 6. The Columns 7 to 9 list the number of reported Single Fault Diagnoses (SFDs), the total number of reported diagnoses, and the sum of the diagnosis cardinality, respectively. The Columns 10 to 13 give the same information when using the conservative mode (Equation 4.9 instead of Equation 4.8). TCAS version 38 contains a wrong array size. FoREnSiC does not detect this bug and thus does not start the diagnosis engine. Finally, the last two columns show the execution time and the number of reported (single-fault) diagnoses when using WPLoc.

Bug-Assist. The tool Bug-Assist [130] computes diagnoses from a counterexample using a solver for maximum satisfiability [131]. See Section 5.3 for a discussion of the working principle of this approach. Just as for FoREnSiC, we used the TCAS reference implementation with an assertion comparing the results as a specification. With only 6.9 seconds on average, Bug-Assist is very fast (Column 4). However, as can be seen from Column 5, it also produces quite a high number of potential fault locations (15 on average)[19].

FoREnSiC in non-conservative mode. With the parameters chosen for Table 4.2 (at most 400 execution paths, $|J| = 3$ input vectors; different parameters will be explored in the next paragraphs), the non-conservative mode is very fast in doing fault localization: it takes only 1.4 seconds on average (Column 6). Note, however, that this number does not include the time for program analysis, which is around 16 seconds. Unfortunately, this configuration produces a rather high number of diagnoses: Our preprocessing identifies between 74 and 79 components in the program, and 34 of them are reported as potential SFDs on the average (Column 7). This number is twice as high as with Bug-Assist. However, Bug-Assist reports its results as potentially faulty lines of code, while our approach often identifies several components per line of code. For cases where the number of single-fault diagnoses is low, our approach often reports a high number of diagnoses with higher cardinality (Column 8 and 9). Since diagnoses of higher cardinality can be considered as less likely, it is important that our approach computes diagnoses in the order of increasing cardinality and that the user can set a bound on the number and the cardinality of diagnoses to compute. One reason for the limited diagnostic resolution is that we only analyzed a small portion of the execution paths in the preprocessed program $\hat{P}$. The program does not contain any loops. Nevertheless, exhaustive program analysis is infeasible: We aborted an experiment to analyze all execution paths of $\hat{P}$ after around $2 \cdot 10^5$ execution paths and 6 hours.

[18] Logical connectives such as `&&` or `||` in `if`-conditions are often swapped. Our front end decomposes `if`-conditions containing such connectives into several `if`-statements. E.g., `if(a && b)` is decomposed into `if(a){if(b){...}}`. The reason is the lazy evaluation semantic in the programming language C, saying that `b` must only be evaluated if `a` is `true`.

[19] We manually subtracted the number of potential fault locations that were reported within in the reference implementation.

Table 4.2: Performance results for fault localization on the TCAS benchmark. Bug-Assist [130] serves as baseline for comparison. Results with FoREnSiC are reported for two modes: the non-conservative mode using Eq. 4.8 and the conservative mode using Eq. 4.9, both with a program analysis limit of 400 execution paths and $|J| = 3$ input vectors. For TCAS version 38, no bug was detected. WPLoc was run with default parameters. The last line lists the average per column. The symbol # is short for "number of".

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			Bug-Assist		FoREnSiC + Eq. 4.8				FoREnSiC + Eq. 4.9				WPLoc	
Version	Matches fault model	# modifications	Execution time	# fault locations	Fault loc. time	# SFDs	# diagnoses	$\sum \lvert\Delta_i\rvert$	Fault loc. time	# SFDs	# diagnoses	$\sum \lvert\Delta_i\rvert$	Execution time	# fault locations
	[-]	[-]	[sec]	[-]	[sec]	[-]	[-]	[-]	[sec]	[-]	[-]	[-]	[sec]	[-]
1	Yes	1	7.8	16	0.7	48	53	58	1.4	3	3	3	20	4
2	Yes	1	9.8	17	0.4	34	91	148	1.4	1	5	9	6.9	2
3	No	1	10	17	1.7	28	367	706	2.7	1	38	119	111	13
4	No	1	7.5	17	1.0	40	64	90	2.0	4	4	4	22	0
5	No	1	4.3	18	5.1	26	382	824	2.1	2	35	92	84	7
6	Yes	1	5.6	17	0.5	44	49	54	1.5	2	6	10	6	2
7	Yes	1	7.6	17	0.9	27	370	713	1.7	1	17	65	6	1
8	Yes	1	8.1	15	1.0	32	337	642	1.6	1	17	65	6.8	1
9	Yes	1	8.6	13	0.5	42	47	52	2.1	2	3	4	20	2
10	Yes	2	11	18	1.5	41	80	119	1.4	1	1	1	6.9	4
11	No	3	6.3	9	4.6	29	356	697	0.4	0	0	0	6.9	4
12	No	1	5.5	18	1.1	41	48	55	1.0	3	14	25	89	5
13	Yes	1	6.7	16	3.3	29	368	707	2.6	3	40	121	111	8
14	Yes	1	7	8	0.1	10	10	10	1.0	1	1	1	104	2
15	No	2	4.4	18	2.3	25	334	643	2.5	1	20	39	35	6
16	Yes	1	7.7	16	1.0	27	370	713	1.6	1	17	65	6.8	1
17	Yes	1	8	16	0.9	27	370	713	1.7	1	17	65	6.8	1
18	Yes	1	7.8	16	1.0	27	370	713	1.7	1	17	65	6.8	1
19	Yes	1	7.9	16	0.9	27	370	713	1.6	1	17	65	7	1
20	Yes	1	7.3	17	0.6	45	50	55	2.5	6	7	8	20	2
21	Yes	1	10	17	0.2	45	47	49	1.5	5	5	5	18.4	2
22	Yes	1	7.1	16	0.1	40	42	44	1.3	1	1	1	18.3	2
23	Yes	1	8.7	13	0.2	42	44	46	1.6	1	1	1	18.4	2
24	Yes	1	10	17	0.2	45	47	49	1.8	5	5	5	18.5	2
25	Yes	1	6.8	16	0.6	46	51	56	1.4	3	3	3	20	3
26	No	1	6.2	17	4.1	28	367	706	2.8	2	39	120	93	8
27	No	1	4.2	18	5.1	26	382	824	2.1	2	35	92	87	7
28	Yes	1	8.2	14	3.1	26	605	1524	1.5	0	11	26	10	2
29	Yes	1	7.4	13	0.5	33	91	149	0.5	0	4	8	5.6	1
30	Yes	1	9.1	17	0.5	15	152	289	0.5	0	4	8	6	2
31	No	3	3.4	15	0.1	35	40	45	1.7	1	41	373	129	10
32	No	3	3.2	17	0.2	34	41	48	0.9	1	1	1	102	7
33	No	4	0.3	1	3.3	32	337	642	1.6	1	17	65	6.8	0
34	No	1	4.3	18	2.2	24	380	822	1.6	2	35	92	30	1
35	Yes	1	10	18	3.0	26	605	1524	1.6	0	11	26	7	2
36	Yes	1	4.3	15	0.7	48	53	58	0.6	0	0	0	129	13
37	No	1	7.4	15	1.0	32	337	642	0.5	1	17	65	6.1	0
38	No	1	0.3	1	-	-	-	-	-	-	-	-	5.2	0
39	Yes	1	7	16	0.6	46	51	56	1.4	3	3	3	20	3
40	No	2	7.6	16	0.4	41	46	51	2.3	0	180	1854	80	8
41	No	1	6.4	16	0.6	42	47	52	1.9	4	4	4	17	3
Avg.	61%	1.3	6.9	15	1.4	34	206	403	1.6	1.7	17	89	37	3.5

Table 4.3: Fault localization results for TCAS with different values of $|J|$. All entries are averages over all TCAS instances. For both the non-conservative mode (Eq. 4.8) and the conservative mode (Eq. 4.9), the number of analyzed execution paths was limited to 400. The symbol # is short for "number of".

	FoREnSiC + Eq. 4.8					FoREnSiC + Eq. 4.9				
$|J|$	1	2	3	5	10	1	2	3	5	10
Avg. # SFDs	45.2	36.9	33.9	32.5	31.7	2.43	1.95	1.70	1.55	1.33
Avg. fault loc. time [sec]	0.16	0.63	1.39	2.72	6.21	0.51	1.10	1.59	3.10	8.51

FoREnSiC in conservative mode. The conservative mode appears to cope with the highly incomplete diagnostic information obtained by our shallow program analysis way better. On average, only 1.7 single-fault diagnoses are reported (Column 11). For those 23 versions where the introduced bug directly corresponds to a set of components identified by FoREnSiC, we investigated the reported diagnoses manually. Only in 17 % of the cases, the expected diagnosis was missed. For 11 cases, the expected diagnosis was reported as the only SFD. In 6 of these 11 cases, all non-single-fault diagnoses had a cardinality of at least 4. Moreover, these excellent results are achieved within very short execution times (Column 10).

WPLoc. With an average execution time of 37 seconds (Column 14), our fault localization approach based on deductive verification is the slowest configuration investigated in Table 4.2. On the positive side, the number of reported SFDs is much lower than with Bug-Assist. Note, however, that WPLoc faces a simpler task: since the specification is given in the form of pre- and postconditions for each function, WPLoc only needs to locate the fault within the functions for which verification fails. In the conservative mode, FoREnSiC still produces fewer SFDs than WPLoc, even though it operates on a global specification. However, while FoREnSiC missed the expected diagnosis in 17 % of the cases, WPLoc did not miss the expected diagnosis in any of the cases where the fault model matches. We therefore consider WPLoc as the configuration with the highest diagnostic resolution in Table 4.2.

Impact of $|J|$. Table 4.3 investigates different trade-offs between execution time and diagnostic resolution that can be achieved with FoREnSiC using different numbers $|J|$ of input vectors that are considered simultaneously (see Section 4.3.2). The first line gives the number of single-fault diagnoses that are reported on the average over all TCAS instances. The second line lists the average execution time. As in Table 4.2, this number does not include the time for program analysis, which is 16 seconds on average. The program analysis depth was again limited to 400 execution paths. The percentage of missed expected diagnoses (17 % when using Equation 4.9 and 4 % when using Equation 4.8) does not change with $|J|$ in Table 4.3. We can observe that using more than one input vector is indeed beneficial for the amount of reported SFDs. However, for larger values of $|J|$, the effect levels out. This experiment suggests that taking only a few input vectors can be a reasonable choice. The execution time increases with $|J|$ but is still low compared to the program analysis time, even for $|J| = 10$.

Impact of the program analysis depth. Table 4.4 summarizes fault localization results with different limits on the number of execution paths to analyze. The first line gives the average number of single-fault diagnoses that are reported. For those 23 TCAS versions where the introduced bug corresponds to components identified by FoREnSiC, we again investigated if the expected diagnosis was reported. The second line gives the percentage of the cases where this expected diagnosis was missed. The last two lines give the average execution time for fault localization and program analysis, respectively. We can observe that the number of reported SFDs is not affected significantly by the number of explored execution paths. The percentage of missed diagnoses increases strongly for low numbers of analyzed execution paths. The main effect here is that for very low numbers, no bug is detected so the diagnosis engine is not even started. If enough paths were analyzed to detect a bug, even significant increases of the number of execution paths do not yield noticeable improvements any more. The execution time for fault localization is very low compared to that for program analysis. This is mostly due to

Table 4.4: Fault localization results for TCAS with different program analysis depths. All entries are averages over all TCAS instances. For both the non-conservative mode (Eq. 4.8) and the conservative mode (Eq. 4.9), the number of $|J|$ of input vectors was set to 3. The symbol # is short for "number of".

	FoREnSiC + Eq. 4.8					FoREnSiC + Eq. 4.9				
# analyzed execution paths	50	100	200	400	4000	50	100	200	400	4000
Avg. # SFDs	35.8	34.8	34.6	33.9	33.8	1.71	1.70	1.72	1.70	1.70
Exp. diagnosis missed [%]	26	17	9	4	4	39	30	22	17	17
Avg. fault loc. time [sec]	0.34	0.48	0.77	1.39	2.26	0.21	0.35	0.69	1.59	4.62
Avg. prog. anal. time [sec]	2.90	4.73	8.49	16.2	243	2.80	4.68	8.49	16.7	261

making use of unsatisfiable cores and incremental solving, as illustrated in the next paragraph.

Impact of exploiting solver features. Figure 4.4 presents a scatter plot that illustrates the speedup we achieve in fault localization on the TCAS benchmark by exploiting features of the underlying SMT solver. The x-axis gives the fault localization time without incremental solving and without using unsatisfiable cores computed by the SMT solver. The y-axis contains the execution time when these features are enabled. Each point in the diagram corresponds to one TCAS instance for a certain number of analyzed execution paths and a certain number $|J|$ of input vectors. We varied these parameters as in Table 4.3 and Table 4.4 to obtain all the data points shown in Figure 4.4.

The blue pluses illustrate the speedup due to exploiting unsatisfiable cores computed by the SMT solver. The average fault localization time is reduced from 146 seconds to only 42 seconds, which corresponds to a speedup factor of around 3.5. The average over the individual speedup factors is even 6.3. We can see that the blue data points are roughly clustered into two clouds. The upper cloud corresponds to the non-conservative mode (using Equation 4.8), the lower cloud to the conservative mode (Equation 4.9). One reason for the higher speedups in the conservative mode can be that it produces more diagnoses of higher cardinality, for which a larger number of conflicts need to be computed and minimized.

The red crosses illustrate the speedup that is achieved using unsatisfiable cores and incremental solving together. The average computation time decreases from 146 seconds to only 2.2 seconds. This is a speedup of a factor of 66. The average over the individual speedup factors is 62.

4.9.3.2 Fault Correction Results

We now investigate the performance of our program repair approach, as implemented in FoREnSiC, on the TCAS benchmark. A comparison of different parameter configurations will then be performed in Section 4.9.3.3. Section 4.9.3.4 will finally compare the performance of FoREnSiC with that of the program sketching tool Sketch [196, 197].

Setup. As a basis for fault correction, we use the two fault localization configurations from Table 4.2 (400 execution paths analyzed, $|J| = 3$ input vectors, one configuration uses Equation 4.8, the other one Equation 4.9). The repair engine is always run in the non-conservative mode, i.e., using Equation 4.8. On-the-fly program analysis is disabled, our heuristic for computing simple repair candidates and nasty counterexamples (see Section 4.5.3) is enabled, and incremental SMT solving is enabled as well. We also perform an additional program analysis pass per diagnosis to repair (see Section 4.8) in order to obtain more accurate diagnostic information. The maximum size of diagnoses to repair is set to 1 (i.e., only single-fault diagnoses are repaired) and the maximum number of repairs to compute per diagnosis is limited to 3. Linear integer arithmetic is used as theory for SMT solving and Z3 is used as solver. We did not impose a limit on the maximum number of repair candidate refinements in the CEGIS loop but set a timeout of 60 seconds for all SMT solver calls. Other configurations will be investigated in Section 4.9.3.3.

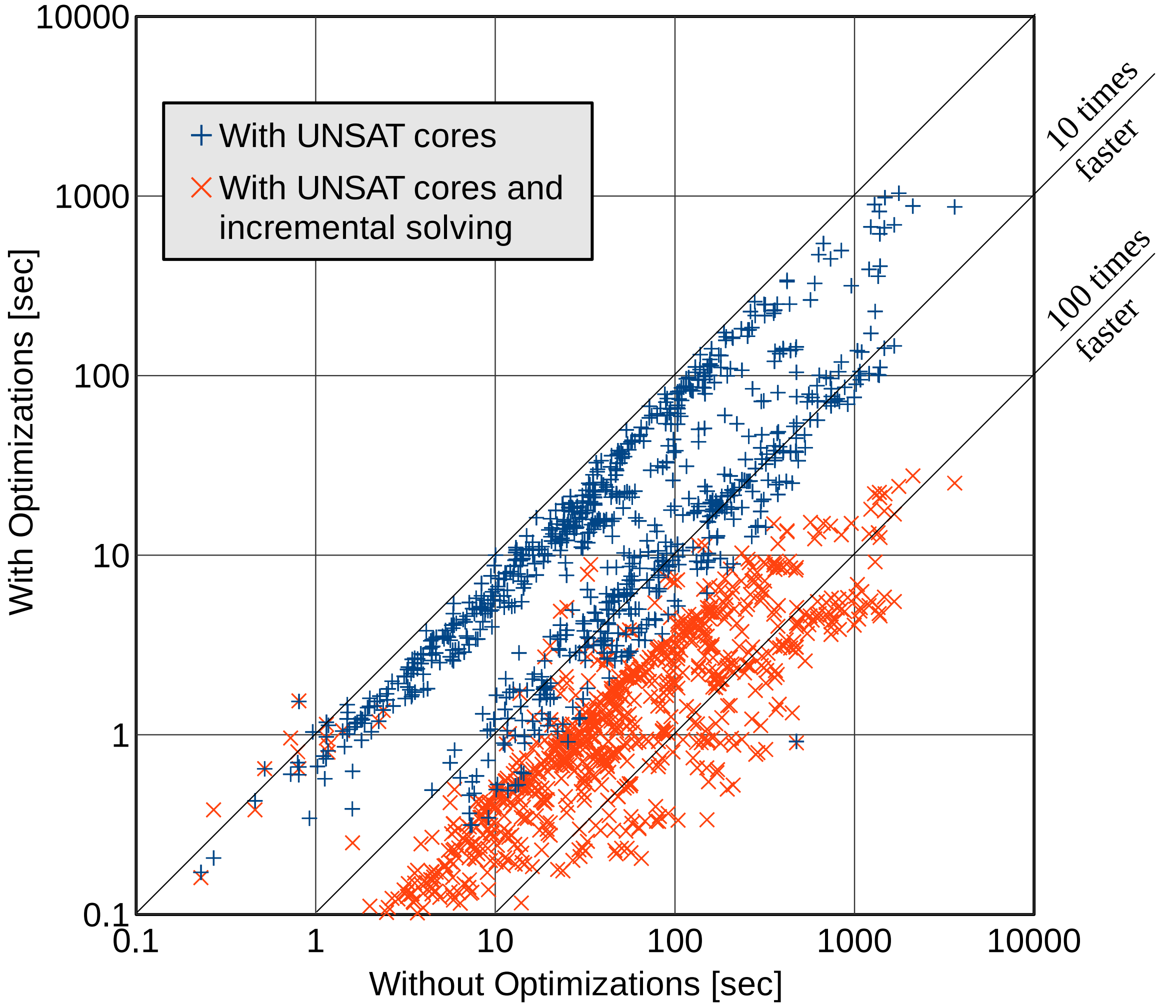


Figure 4.4: A scatter plot illustrating the speedup due to exploiting solver features in fault localization on the TCAS benchmark. The x-axis contains the fault localization times without incremental solving and without using unsatisfiable cores computed by the SMT solver. The y-axis gives the execution time when unsatisfiable cores (blue pluses) and incremental solving (red crosses) are enabled. Note the logarithmic scale on both axes.

Table of results. Table 4.5 summarizes performance results in this setup for the 41 faulty versions of the TCAS benchmark. Just like in Table 4.2, Column 1 lists the TCAS version number and Column 2 indicates whether the bug in the respective version matches our fault model of incorrect expressions. The Columns 3 to 7 give fault correction statistics when using the fault localization setting from Table 4.2 in the non-conservative mode (i.e., using Equation 4.8). The number of single-fault diagnoses (SFDs) that are used as a basis for computing repairs is shown in Column 3. Column 4 gives the total time spent on fault correction, including the time for additional program analysis passes. Column 5 lists the total number of reported repairs. We used the model checker CBMC [61] to check if the computed repairs are indeed equivalent to the reference implementation. Column 6 gives the number of repairs for which this was the case. Column 7 finally lists the time spent by the repair engine until the first repair is reported. The Columns 8 to 12 give the same information but with diagnoses computed in the conservative mode (using Equation 4.9).

Missed repairs. Even if the fault model matches, FoREnSiC is unable to find a repair in 5 cases (compare Column 2 and 5). For the TCAS versions number 22, 23 and 24, the reason is that the faulty expression is missing a call to a (side-effect-free) function. However, we only define our templates as expressions over all program variables that are in scope. An extension to also include calls to side-

Table 4.5: Performance results for fault correction with FoREnSiC on the TCAS benchmark. FoREnSiC was run with the two fault localization settings from Table 4.2, i.e., using Eq. 4.8 and Eq. 4.9. The repair engine was run using Eq. 4.8 in both cases. The maximum size of diagnoses to repair was set to 1, the maximum number of repairs to compute per diagnosis was set 3. The symbol # is short for "number of". The last line lists the average per column.

1	2	3	4	5	6	7	8	9	10	11	12
		FoREnSiC + Eq. 4.8					FoREnSiC + Eq. 4.9				
Version	Matches fault model	# SFDs	Fault corr. time	# repairs	# correct repairs	Time until first repair	# SFDs	Fault corr. time	# repairs	# correct repairs	Time until first repair
	[-]	[-]	[sec]	[-]	[-]	[sec]	[-]	[sec]	[-]	[-]	[sec]
1	Yes	48	3078	16	16	799	3	275	3	3	223
2	Yes	34	839	3	3	91	1	57	3	3	16
3	No	28	1669	0	0	-	1	15	0	0	-
4	No	40	2377	19	18	44	4	105	6	6	23
5	No	26	1172	0	0	-	2	30	0	0	-
6	Yes	44	2290	19	3	273	2	39	3	2	39
7	Yes	27	545	3	3	7	1	8	3	3	8
8	Yes	32	1203	19	19	8	1	8	3	3	8
9	Yes	42	1480	3	3	209	2	83	3	3	83
10	Yes	41	2588	8	8	583	1	118	0	0	118
11	No	29	731	0	0	-	0	-	-	-	-
12	No	41	1321	0	0	-	3	73	0	0	-
13	Yes	29	1355	3	3	290	3	138	3	3	41
14	Yes	10	492	3	3	68	1	57	3	3	36
15	No	25	961	0	0	-	1	18	0	0	-
16	Yes	27	993	31	3	7	1	7	3	3	7
17	Yes	27	525	3	3	7	1	8	3	3	8
18	Yes	27	619	6	3	8	1	9	3	3	8
19	Yes	27	937	13	3	7	1	7	3	3	7
20	Yes	45	2687	18	18	112	6	447	3	3	107
21	Yes	45	2718	10	10	2417	5	473	0	0	473
22	Yes	40	1173	0	0	-	1	78	0	0	-
23	Yes	42	1369	0	0	-	1	145	0	0	-
24	Yes	45	2711	0	0	-	5	405	0	0	-
25	Yes	46	1924	3	3	915	3	121	3	3	111
26	No	28	1390	0	0	-	2	107	0	0	-
27	No	26	1153	0	0	-	2	32	0	0	-
28	Yes	26	530	2	2	8	0	-	-	-	-
29	Yes	33	647	0	0	-	0	-	-	-	-
30	Yes	15	202	0	0	-	0	-	-	-	-
31	No	35	1943	19	19	188	1	13	3	3	13
32	No	34	1924	22	22	308	1	13	3	3	13
33	No	32	794	0	0	-	1	16	0	0	-
34	No	24	872	3	2	562	2	40	0	0	40
35	Yes	26	523	2	2	11	0	-	-	-	-
36	Yes	48	939	1	1	939	0	-	-	-	-
37	No	32	847	0	0	-	1	136	0	0	-
38	No	-	-	-	-	-	-	-	-	-	-
39	Yes	46	1888	3	3	916	3	116	3	3	109
40	No	41	1098	15	15	219	0	-	-	-	-
41	No	42	2685	20	20	180	4	123	6	6	22
Avg.	61%	34	1380	6.7	5.2	526	1.7	101	1.9	1.9	69

effect-free functions would be possible[20], but increases the search space for repairs and thereby the potential computational effort for finding correct repairs. For the TCAS versions 29 and 30, the faulty expression is not identified as a component in the preprocessing phase. Recall from Section 4.9.1.1 that our implementation only considers the conditions and the right-hand sides of assignments as potentially faulty components, relying on the front end to assign non-trivial expressions to fresh variables before they are used. This approach works well in most cases, but fails for these two versions. The version with conservative fault localization fails to find repairs for some more versions where the fault model matches (compare Column 2 and 10). The reason lies in diagnoses being missed by the fault localization step.

Unexpected repairs. For 6 TCAS versions where the fault model does not match, FoREnSiC is able to find repairs nevertheless. As an example, we discuss Version 41, which contains the following code.

```
1  int Non_Crossing_Biased_Climb() {
2    int result = 0;
3    if (Inhibit_Biased_Climb() > Down_Separation) {
4      result = [code skipped]
5    } else {
6      // deleted: if(Own_Above_Threat())
7      if (Cur_Vertical_Sep >= 300)
8        if (Up_Separation >= ALIM())
9          result = 1;
10   }
11   return result;
12 }
```

The deletion of Line 6 makes the function `Non_Crossing_Biased_Climb` return 1 in cases where only 0 would render the entire TCAS program correct. One repair suggested by FoREnSiC is to modify the condition in Line 7 to `0 != 0`, i.e., to false. This is surprising, because it makes the entire `else`-part obsolete. The reason why this works is that the function `Non_Crossing_Biased_Climb` is only used in a statement

```
need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
```

in some other function. That is, `need_upward_RA` can only be 1 if `Own_Below_Threat()` returns 1. Yet, if `Own_Below_Threat()` gives 1, then `Own_Above_Threat()` necessarily returns 0. Hence, the `else`-part in the reference implementation is indeed obsolete and can be removed. FoREnSiC finds 19 more repairs for the problem. Some are similar in spirit to the discussed fix. Other repairs modify the computation or the use of `need_upward_RA`. For other TCAS versions, the author of this thesis often failed to comprehend why certain repairs produced by FoREnSiC render the program correct (but correctness was confirmed by the model checker CBMC [61]). This illustrates that FoREnSiC can produce non-trivial fixes that may not be obvious for the user.

Correctness of repairs. We verified the computed repairs using the model checker CBMC [61]. With the diagnoses computed in the non-conservative mode (using Equation 4.8), 208 of the 267 reported repairs were found to be correct (compare Column 5 and 6 in Table 4.5). With the diagnoses computed in conservative mode (using Equation 4.9), only one of the 63 computed repairs was incorrect (Column 10 versus 11). The reason for the incorrect repairs is that we run the repair engine in the non-conservative mode (using Equation 4.8) but limited the program analysis depth to only 400 execution paths. When we increase the number to 2000 execution paths, only correct repairs are reported. The average execution time per instance increases from 1380 seconds to 2175 seconds in Column 4, and from 101 to 157 seconds in Column 9. This illustrates how our approach can trade performance for accuracy.

[20] For instance, for each side-effect-free function `foo(p1, ..., pn)` we could add the code block `int p1 = 0; ... int pn = 0; int rfoo = foo(p1, ..., pn);` at the beginning of each function during the preprocessing phase. The repair engine can then use the variable `rfoo` in repair templates. It can also repair assignments to the parameter variables `p1`, ..., `pn` to find appropriate arguments for the function call.

Table 4.6: Performance results for fault correction on the TCAS benchmark with FoREnSiC in different configurations. All configurations had a program analysis depth limit of 2000 execution paths. The given numbers are averages over 26 TCAS instances. The symbol # is short for "number of".

	Number	1	2	3	4	5	6	7
Configuration	Program analysis	2x	2x	2x	2x	1x	otf	otf
	Candidate verification	SMT	SMT	SMT	SMT	SMT	test	test
	Conservative	No	Yes	No	No	No	No	No
	Heuristics	Yes	Yes	Yes	No	Yes	Yes	Yes
	Incremental solving	Yes	Yes	No	Yes	Yes	Yes	No
Results	# correct repairs	26	24	26	15	4	26	26
	Avg. fault corr. time [sec]	22.8	35.2	44.4	101	12.9	10.3	10.8
	Avg. # iterations	6.73	9.76	6.88	621	3.42	5.58	5.77
	Avg. cand. comp. time [sec]	0.37	11.7	21.8	53.7	7.84	0.04	0.44
	Avg. cand. verif. time [sec]	0.21	0.59	0.21	17.0	0.13	1.63	1.58

Execution time. With the many diagnoses computed using Equation 4.8, fault correction already takes around 25 minutes per TCAS instance on average (Column 4). One reason is also that we asked the repair engine to compute 3 diagnoses per repair. The average time until the first repair is reported is less than 10 minutes (Column 7). With the smaller amount of diagnoses found using Equation 4.9, the repair engine takes only 2 minutes on average (Column 9). Compared to the time for program analysis (16 seconds on average) and fault localization (only a few seconds), we can thus conclude that fault correction dominates the total debugging time, which is not surprising. In any case, even though the TCAS benchmark contains logic that is all but trivial, the total debugging times are still acceptably low.

4.9.3.3 Comparison of Different Fault Correction Configurations

In this section, we evaluate and compare the performance of FoREnSiC's repair engine in different configurations. The performance results in Table 4.5 are strongly influenced by the accuracy of the diagnoses reported by the preceding fault localization step. In order to eliminate this effect, we now take only the 26 TCAS instances for which a repair has been found in Table 4.5 (thus we know that a repair exists). For these instances, we let FoREnSiC compute one repair for one diagnosis that has a solution.

Results. Table 4.6 summarizes the results. The upper six rows describe the compared configurations, the lower five rows summarize the corresponding performance results. The first row assigns a unique number to each configuration. The program analysis entry "2x" means that an additional program analysis pass (see Section 4.8) for the diagnoses is performed to obtain more accurate diagnostic information, "1x" means that this pass is skipped, and "otf" means that program analysis is done on the fly (Section 4.7). Candidate verification is either done by computing a counterexample using an SMT solver call on the negated correctness formula (entry "SMT") or by executing the 1608 test cases (entry "test") for the TCAS program. The fourth row indicates whether the repair engine was run in conservative mode (using Equation 4.9) or in the non-conservative mode (using Equation 4.8). The row labeled by "Heuristics" indicates if our heuristics for computing simple candidates and nasty counterexamples (Section 4.5.3) are enabled. The last row describing the configuration indicates whether incremental SMT solving is enabled or disabled. In all configurations, the program analysis depth was limited to 2000 execution paths. The rows in the lower part of Table 4.6 summarize the number of benchmark instances for which a repair was found, the total fault correction time (including the execution time for additional program analysis passes) on the average over the instances, the average number of iterations of the CEGIS loop, and the average (over the benchmark instances) of the total time for computing repair candidates and for verifying them, respectively.

Discussion. Configuration 1 serves as baseline for our investigation. It is essentially the configuration that is also used in Table 4.5 but with the difference that we analyzed up to 2000 execution paths instead of 400 to avoid incorrect repairs.

Conservative mode. When using the conservative mode (Configuration 2), the average time for repair candidate computation increases significantly (from 0.37 to 11.7 seconds) and we even hit a timeout for one case. One reason is that concolic execution usually finds way more execution paths that satisfy the specification than ones violating the specification. Consequently, the correctness formula computed with Equation 4.9 is often much larger than with Equation 4.8. For a second TCAS instance, FoREnSiC concludes that no repair exists, which means that analyzing 2000 execution paths was still not enough when using Equation 4.9.

Incremental solving. When disabling incremental SMT solving (Configuration 3 versus 1), the average repair candidate computation time increases even more significantly (from 0.37 to 21.8 seconds). That is, the effect of incremental SMT solving to candidate computation (the only place where it is applied) is enormous. Even though the total fault correction time is also significantly influenced by the time for the additional program analysis pass (which takes around 22 seconds on average), incremental solving in the repair candidate computation still halves the total fault correction time.

Heuristics. When disabling our heuristics (Configuration 4 versus 1), the number of iterations in the CEGIS loop grows by around two orders or magnitude (from 6.73 to 621) on average in our experiments. Since the formula for repair candidate computation grows in every iteration (see Algorithm 4.2), this results in timeouts for quite some instances. In 19 cases, a repair was still reported, but only 15 of the reported repairs were correct according to CBMC. The reason for incorrectness was mostly in variable overflows due to very high template parameter values (recall that we used linear integer arithmetic, which disregards overflows, for computing repairs). The fact that so many repairs are still found is also due to incremental SMT solving in the candidate computation. When disabling incremental solving as well, five more cases end in a solver timeout.

Second program analysis pass. When disabling the second program analysis pass (Configuration 5 versus 1), FoREnSiC reports 23 repairs rather quickly. However, only 4 of them are equivalent to the reference implementation according to CBMC. The reason is that analyzing 2000 execution paths in the preprocessed program $\hat{P}$ yields only rather imprecise diagnostic information for the particular diagnoses at hand. Analyzing the preprocessed program where only the diagnosis to repair is left open gives much more accurate information with the same number of analyzed program paths. The obvious disadvantage is that the additional program analysis pass also consumes some computation time.

On-the-fly program analysis. Our approach with on-the-fly program analysis and test case execution for verifying candidates (Configuration 6) takes only half the execution time of Configuration 1 and still finds repairs for all cases. This has several reasons. First, less time is spent on program analysis. On average, on-the-fly program analysis investigated only 15.2 execution paths. In contrast, Configuration 1 analyzed 653 execution paths in the second program analysis pass on average. Second, repair candidate computation is extremely fast because the correctness formula that is passed to the SMT solver contains only those aspects of the program behavior that are needed by CEGIS. Even when incremental solving is disabled (Configuration 7), the average repair candidate computation time is still below one second.

Workload distribution in CEGIS. For those configurations in Table 4.6 where candidate verification is done with an SMT solver call on the negated correctness formula, repair candidate computation usually takes significantly more computation time than repair candidate verification in the CEGIS loop (compare the last two rows in Table 4.6). This is not surprising because the SMT formula for candidate computation grows with every iteration, while the formula for candidate verification is (rather) constant in size (see Algorithm 4.2). Only in our configurations with test case execution as a means for candidate verification, the verification times are higher. Because the TCAS program is rather small, the reason is mostly the overhead for compiling candidate programs and executing the test cases in a separate process. For larger programs, this overhead can be expected to be negligible compared to program analysis and candidate computation, though.

4.9.3.4 Comparison with Sketch

In this section, we compare the performance of our repair engine to Sketch [196, 197], which is a tool for program sketching: The user provides a program with unknown integer values (so-called "holes") together with a specification. Sketch then synthesizes values for the unknown integer holes using CEGIS such that the specification is fulfilled for all input values. For synthesizing more complex program parts, the user has to provide so-called "generators", which are expressions containing only unknown integer values. These generators serve the same purpose as our repair templates: reducing the synthesis of program components to the search for integer constants. Sketch models the semantics of the program under analysis on the propositional level and uses a SAT solver to realize the CEGIS loop. In contrast, our repair approach uses an SMT solver, applies additional heuristics, and can trade accuracy for efficiency using many parameters.

Setup. We modeled the 26 TCAS instances from Table 4.6 in the input language of Sketch using the reference implementation as a specification. Next, we manually replaced the faulty components that were also used as basis for repair in Table 4.6 with templates for new expressions, where the unknown template parameters were modeled using holes. Finally, Sketch was called to synthesize template parameter values. Our experiments were performed using Sketch version 1.6.7 (released in September 2014) with default parameters and MiniSat as the underlying solver. We set a memory limit of 4 GB.

Results. Table 4.7 summarizes the performance results. The first column lists the TCAS version number. The second column gives the number of template parameters used by FoREnSiC. Column 3 gives the repair synthesis time with FoREnSiC in Configuration 1 from Table 4.6. In order to have Sketch find a repair with a 4 GB memory limit, we had to reduce the bit-width of an integer to 8, for which we had to lower constants in the program. We also had to manually reduce the number of template parameters. The number of parameters for which Sketch can still find a repair without running out of memory is listed in Column 4. The last column gives the corresponding execution time with Sketch.

Discussion. The memory consumption of Sketch on the TCAS benchmarks is very high. Even though we reduced the bit-width of integers, the same template as in FoREnSiC could only be used in one case (Version 34). For all other instances, we had to reduce the number of template parameters significantly (from 15.3 to 4.6 on average) in order to find a repair without exceeding the memory limit of 4 GB. On the other hand, the maximum memory consumption of FoREnSiC was below 100 MB in all cases. The execution time of Sketch is also higher by one order of magnitude on the average over our benchmarks, even though the templates for Sketch were reduced. We attribute these enormous performance differences mostly to the fact that Sketch models the program semantics on the propositional level while our approach is based on SMT solving (with linear integer arithmetic in this experiment).

4.9.4 Discussion

Our approach for applying controller synthesis techniques in the application of automatic program repair turned out to be promising in various respects.

Usefulness. As demonstrated on several examples in Section 4.9.2, our debugging flow from Figure 4.1 can deliver helpful diagnostic information to the user. Our key ingredients to achieve this include a generic and fine-grained fault model that can pinpoint potential reasons for mismatches between the program and its specification precisely, systematic fault localization using model-based diagnosis to filter out promising candidates for repair and, ultimately, our template-based approach to synthesize repairs that are not only correct (with configurable approximations) but also simple and understandable by humans. Our proof-of-concept implementation in FoREnSiC is fully automatic and easy to apply.

Scalability. Our approach does not only work for toy examples but also scales to non-trivial programs for which manual debugging can be difficult and time-consuming. One success factor here is that our approach provides many parameters to trade efficiency for accuracy. This includes our concept for incomplete program analysis using symbolic or concolic execution and user-given bounds on the

Table 4.7: Performance comparison with Sketch [196, 197] for the TCAS instances from Table 4.6. FoREnSiC was run with Configuration 1 from Table 4.6. Sketch version 1.6.7 was run with MiniSat and default parameters. We set a memory limit of 4 GB and reduced the bit width of integers in Sketch to 8 in order to prevent Sketch from running out of memory. The number of template parameters had to be reduced as well. The symbol # is short for "number of". The last line gives the averages over the columns.

	FoREnSiC		Sketch (8 bit)	
Version	# Parameters	Time	# Parameters	Time
	[-]	[sec]	[-]	[sec]
1	16	12	3	5.7
2	13	20	3	1.6
4	14	7	6	2.5
6	14	73	3	56
7	13	6	3	418
8	13	7	3	250
9	15	13	2	4.4
10	28	130	4	1777
13	14	21	3	207
14	14	30	3	6.4
16	13	6	3	103
17	13	6	3	431
18	13	6	3	1091
19	13	7	3	232
20	15	12	4	7.1
21	20	23	4	12
25	16	10	3	5.6
28	14	25	10	996
31	18	11	4	166
32	18	10	4	163
34	17	98	17	24
35	14	25	10	1000
36	15	6	4	450
39	16	11	3	536
40	14	10	6	1308
41	15	7	6	1186
Avg.	15.3	22.8	4.6	401

analysis depths, replacing universal quantifications with finite conjunctions in fault localization, and our approach of repair synthesis with on-the-fly program analysis to obtain more focused information about the program correctness under specific inputs. Another success factor is the utilization of SMT solvers as underlying reasoning engines. Our comparison with Sketch showed that handling the problem on the propositional level using SAT solvers can be significantly more expensive in the domain of software programs. Using BDDs instead of SAT solvers can be expected to scale even much worse due to the very high number propositional variables needed to encode the behavior of software programs of reasonable size. With performance improvements of up to two orders of magnitude (see, e.g., Figure 4.4), careful utilization of SMT solver features such as the computation of unsatisfiable cores and incremental solving also turned out to be very beneficial.

Conclusion. We conclude that automatic program repair is a promising application for satisfiability-based controller synthesis techniques. When designed carefully, synthesis can be scalable enough to repair incorrect programs of reasonable size, delivering helpful diagnostic information to the user.

5 Related Work

Parts of this chapter are based on the previous publications of the author on which this thesis is based [82, 39, 31, 140, 141, 145, 30].

Related work on which this thesis builds has already been discussed throughout the document, and especially in Chapter 2. This chapter discusses alternative approaches and points out similarities and differences. Sections 5.1 and 5.2 discuss alternatives to our hardware synthesis algorithms from Chapter 3. Sections 5.3 and 5.4 discuss alternative fault localization and repair approaches and relate them to our software repair flow from Chapter 4. Since automatic synthesis and debugging are wide and active research areas, our discussion will not be exhaustive but focuses on closely related works.

5.1 SAT-Based Hardware Synthesis Approaches

While reactive hardware synthesis is a broad research area, hardware synthesis approaches based on decision procedures for satisfiability are relatively rare.

Incremental induction. Morgenstern et al. [163] present a SAT solver based synthesis algorithm for safety specifications that is inspired by the model checking algorithm IC3 [41] and its principle of incremental induction. The basic idea is to lazily compute the *rank* of the initial state of the specification, which is the maximum number of steps in which the environment can enforce to visit an unsafe state. If this rank is found to be finite, the specification is unrealizable. If it is found to be infinite, the specification is realizable. Hence, strictly speaking, the paper only presents a decision procedure for realizability. However, computing a winning strategy and a circuit implementing this strategy is also possible. We used a reimplementation of this algorithm as a baseline in our experimental evaluation. It was very fast on certain benchmark instances, but outperformed significantly by our new algorithms on average.

Strategy computation without preimages. Narodytska et al. [166] propose an algorithm to compute strategies for reachability specifications, where a set of target states needs to be visited at least once. The general idea is to apply a counterexample-guided backtracking search in order to find a set of executions that is sufficient to reach the target states within some number n of steps. This set of executions is then generalized into a winning strategy in the form of a tree that defines control actions based on previous inputs. If no strategy is found for a particular bound n, then n is increased. A SAT solver is used both to compute and to generalize executions. Hence, in comparison to our work, this approach operates on a different specification class (reachability rather than safety), and it computes a winning strategy directly rather than deriving it from a winning region.

Implementing strategy trees. Eén et al. [78] complete the work discussed in the previous paragraph by proposing a method to compute circuits implementing the obtained winning strategies. Just like one of our methods, it uses interpolation. However, since the strategies are represented as trees rather than relations, the use of interpolation is quite different compared to our work.

QBF-based approaches. Staber and Bloem [201] present a QBF-based synthesis method for safety specifications. The general principle of unrolling the transition relation has already been discussed along with its drawbacks in Section 3.1.1.1 as a motivation for our learning-based algorithms. A solution for Büchi objectives (where some set of states needs to be visited infinitely often, see Section 2.5.4) is presented by Staber and Bloem [201] as well. Alur et al. [5] propose a similar solution for bounded reachability specifications (where a set of target states needs to be reached within at most n steps). This paper [5] also proposes an optimization that uses only one copy of the transition relation. However, all variables are still copied for all time steps and the high number of quantifier alternations (linear in n) remains. In contrast, our learning-based methods use only one copy of the transition relation and two quantifier alternations in all QBF solver calls (at the cost of a potentially higher number of solver calls).

ALLQBF solving. Becker et al. [15] explain how QBF solvers can be used to compute not only one but *all* satisfying assignments of a QBF in the form of a compact (quantifier-free) formula. Similar to some of our satisfiability-based synthesis methods, query learning is used to solve this problem. The paper also points out that such an ALLQBF engine can be used as a direct replacement of BDDs to compute the winning region of various specification classes using fixpoint algorithms. For instance, Algorithm 2.1 can be realized with an ALLQBF engine in order to compute the winning region of a safety specification. While our QBF-based algorithm QBFWIN (Algorithm 3.1) is similar in spirit, there are also some important differences. We apply query learning directly to the specification rather than the preimage computations, which allows for better generalizations. Furthermore, we extend the basic algorithm with additional optimizations such as our reachability optimization from Section 3.1.4.

QBF as a game. Synthesis can be seen as a game between two players: the system controlling the outputs and trying to satisfy the specification, and the environment controlling the inputs and trying to violate the specification. Similarly, QBF solving can also be seen as a game between two players: one player controls the existentially quantified variables and tries to satisfy the formula, the other player controls the universal variables and tries to falsify the formula. This idea is followed by Janota et al. [120] in the QBF solver RAReQS. Following the principle of counterexample-guided refinement of solution candidates, it uses two competing SAT solvers to build a QBF solver: one SAT solver computes candidates in the form of assignments to existential variables, the other one refutes them with assignments for the universal variables. We followed the same principle when traversing from our QBF-based synthesis algorithm to SAT solver based algorithms (cp. Algorithm 3.1 with Algorithm 3.4). However, we apply the idea on the level of the synthesis algorithm rather than for realizing individual QBF solver calls. This allows for additional optimizations. Another connection to this work is in coming to the same conclusion, namely that solving quantified problems with SAT solvers instead of QBF solvers can be beneficial.

SMT-based bounded synthesis. Bounded synthesis [87] by Finkbeiner and Schewe has the objective of synthesizing a reactive system from a given Linear Temporal Logic (LTL) [175] specification. First, the LTL specification φ is transformed into a (universal co-Büchi tree) automaton. A given system implementation satisfies φ if there exists a special annotation that maps each (automaton state, system state)-pair to a natural number. The idea is now to search for such an annotation and a system implementation simultaneously using an SMT solver: An upper bound on the system size is fixed but the system behavior is left open by using uninterpreted functions for the transition relation and the definition of the system outputs. Along with the annotations, the SMT solver then searches for concrete realizations of these uninterpreted functions. In case of unsatisfiability, the bound on the system size is increased until a solution is found. Although this synthesis approach is also SAT-based, it is quite different from the algorithms presented in this thesis. The basic philosophy of enumerating constraints that have to be satisfied by the final solution is similar to our template-based approach and our reduction to EPR, though.

Parameterized synthesis. The tool PARTY [134] uses SMT-based bounded synthesis to solve the parameterized synthesis problem [116], which asks to synthesize systems with a parametric number of isomorphic components. The approach is based on so-called *cutoffs* [83], saying that the verification of parametric systems with an arbitrary number of isomorphic components can be reduced to the verification of systems with a fixed size (the cutoff size) if the specification has a certain structure.

Controller synthesis using uninterpreted functions. Hofferek et al. [111, 113, 110] present an approach to synthesize controllers for aspects that are hard to engineer in concurrent systems. A sequential reference implementation acts as a specification. Uninterpreted functions are used to abstract complex datapath elements. Interpolation over SMT formulas is used as the core technology for computing a controller implementation. This includes a method to compute multiple interpolants from a single unsatisfiability proof [113]. The approach is implemented in the tool Suraq [112]. While there are similarities with our interpolation-based algorithms, we apply interpolation on the propositional level, we do not use abstraction using uninterpreted functions, and we compute one interpolant after the other. These differences appear to be interesting directions for future work, though.

5.2 Other Hardware Synthesis Approaches and Tools

BDDs can be considered as the dominant data structure for symbolic synthesis algorithms. However, there are also other alternatives.

Antichains. Given a set of partially ordered elements, an *antichain* is a subset of elements that are all pairwise incomparable. Just like BDDs, antichains can be used as compact representations of large state sets: for a given partial order among states, an antichain represents the set of all states that are less than or equal to one antichain element with respect to the partial order. Besides decision procedures for satisfiability, antichains provide another successful alternative to BDDs in synthesis [86, 181, 18]. The following paragraphs describe such approaches in more detail.

Antichains for LTL synthesis. Filiot et al. [86] present a synthesis approach for LTL specifications that uses antichains as data structure. It translates the specification into a (universal co-Büchi word) automaton and enforces that the rejecting states of the automaton are visited at most n times. This effectively gives a safety game and is thus similar to bounded synthesis [87] as discussed earlier. The approach has been implemented in the tool Acacia+ [40]. While the similarities to our work are small, the procedure of reducing LTL specifications to safety games can be used to apply our SAT-based synthesis methods also to LTL specifications. In fact, this approach was followed in the SyntComp competition to translate LTL benchmarks into safety specifications automatically [117].

Antichains for synthesis with imperfect information. In certain settings, the system to be synthesized may not be able to observe all internals of other components. Synthesis algorithms for imperfect information address this issue. Raskin et al. [181] present algorithms to determine the realizability of such synthesis problems using antichains. Berwanger et al. [18] extend this work by proposing a method to also extract winning strategies for parity games with imperfect information. This approach has been implemented in the tool Alpaga [17]. As an optimization, this tool uses BDDs to represent antichains in such a way that efficient quantification is possible.

Explicit representations. The tool Lily [125] synthesizes reactive systems from LTL specifications by a serious of automata transformations that are based on work by Kupferman and Vardi [149].[1] A witness to the non-emptiness of the final (nondeterministic Büchi tree) automaton constitutes an implementation of the original specification. Jobstmann and Bloem [125] present a multitude of optimizations to improve the performance of this approach. Lily implements them on top of Wring [199]. Lily does not represent automata symbolically but operates on explicit representations. The similarities to our SAT-based synthesis algorithms are thus rather small.

BDD-based tools. We only give a brief and incomplete overview of BDD-based synthesis tools and approaches. Anzu [126] is a BDD-based synthesis tool for GR(1) specifications [36]. It has later been reimplemented in the requirement analysis tool Ratsy [29]. The same synthesis algorithm is also implemented in the BDD-based tools slugs[2], gr1c[3], and NuGAT[4], which is a game solver built on top of the model checker NuSMV [58]. Unbeast [81] is tool for synthesis from LTL specifications that also builds on the principle of bounded synthesis [87]. The reduction from LTL to safety games is similar to that by Filiot et al. [86] but the resulting safety game is solved using BDDs instead of antichains. Except for our own submission Demiurge, all tools that competed in the SyntComp 2014 competition [117] are BDD-based. This includes AbsSynthe [43], which has been used as a baseline for comparison in our experimental results, Basil by Rüdiger Ehlers, realizer by Leander Tentrup, and the Simple BDD Solver[5] by Leonid Ryzhyk and Adam Walker.

[1] Similar to the antichain-based approach by Filiot et al. [86] and the bounded synthesis approach by Finkbeiner and Schewe [87], the LTL specification is translated into a universal universal co-Büchi tree automaton first. Following an approach by Kupferman and Vardi [149], this automaton is then translated into an alternating weak tree automaton and further on to a nondeterministic Büchi tree automaton.

[2] https://github.com/LTLMoP/slugs (last visit on 2015-08-01).

[3] http://slivingston.github.io/gr1c/ (last visit on 2015-08-01).

[4] http://es.fbk.eu/technologies/nugat-game-solver/ (last visit on 2015-08-01).

[5] https://github.com/adamwalker/syntcomp/ (last visit on 2015-08-01).

5.3 Fault Localization

Techniques for fault localization can roughly be classified into statistical approaches and logic-based approaches. Since our solution for fault localization in the context of repair synthesis falls into the latter category, we will only give a brief overview of statistical methods. Existing logic-based methods will then be discussed in more detail.

Statistical fault localization. The basic idea of statistical fault localization is to collect data about passing and failing executions of a program, and to analyze this data in order to rank program parts according to their "suspiciousness" [150]. Intuitively, if a program statement has been executed in many failing test cases and in only a few passing test cases, then it can be considered as highly suspicious of being responsible for the failures. This principle is also referred to as *spectrum-based* fault localization [184], with a program spectrum being a profile that indicates which parts of the program have been active during an execution. There exists a rich body of literature proposing and comparing various suspiciousness measures. We refer the interested reader to a recent publication by Landsberg et al. [150], which compares 157 different measures experimentally. Existing tools include Tarantula [129] (for C programs), AMPLE [70] (for Java), Zoltar [121] (building on the LLVM compiler infrastructure), and Pinpoint [56] (targeting Internet services). While our model-based fault localization approach is fundamentally different, a combination with statistical techniques is a promising direction for future work.

Model-based diagnosis. Model-based diagnosis [182, 72] has already been applied to locate faults in logic programs [65], functional programs [202], VHDL designs [91], Java programs [158], knowledge bases [85], ontologies [90], temporal logic specifications [143], and spreadsheets [119, 1]. We apply model-based diagnosis to correctness formulas obtained from software programs using symbolic or concolic execution and combine it with automatic program repair. Moreover, we use a customized definition of conflicts and diagnoses for our setting.

Using a model checker. Griesmayer et al. [105, 106] show how diagnoses for an incorrect hardware or software program P can be computed using an off-the-shelf model checker. Our fault localization approach is inspired by this work, so we discuss it in more detail. Assume that the model checker produced a counterexample demonstrating that P does not satisfy a given specification φ. Then, (1) the original inputs of the program are fixed to the values defined by the counterexample, (2) the program is split into components and every component c_i is textually replaced by `if` ab_i `then` x_i `else` c_i, where ab_i and x_i are new inputs, and (3) a model checker verifies the so preprocessed program against the specification $\varphi' = \neg\big(\varphi \wedge ((\sum \mathsf{ab}_i) = 1)\big)$. This way, the model checker searches for values of all ab_i and x_i to falsify φ', i.e., to satisfy the original specification φ while having only one ab_i set to true. If such a counterexample is found, the component c_i for which ab_i is assigned true is a diagnosis because the alternative value x_i fixes the counterexample. Further diagnoses can be computed by refining φ' with the conjunct $\mathsf{ab}_i = \mathsf{false}$ and starting the procedure again. Multiple counterexamples can be used by invoking the procedure for one counterexample after the other and then intersecting the diagnoses. Diagnoses of higher cardinality can be computed by relaxing the cardinality constraint $(\sum \mathsf{ab}_i) = 1$.

This approach has many similarities to our solution. For software programs, it uses the same fault model of incorrect expressions. The preprocessing is also similar in spirit but our approach accounts for program components to behave abnormally during the program analysis with symbolic or concolic execution rather than expressing this possibility with additional `if`-statements in the program code.[6] The definition of a diagnosis via repairability is similar, but our approach considers multiple input vectors simultaneously. The computation of diagnoses is completely different. Griesmayer et al. compute diagnoses directly using a model checker as a black box. Our approach uses the hitting set tree algorithm by Reiter [182] to infer diagnoses from conflicts. This has the advantage that solver features such as incremental solving and the computation of unsatisfiable cores can be exploited effectively. Furthermore, our

[6] Introducing such additional `if`-statements would increase the number of execution paths and thus amplify the path explosion problem of symbolic or concolic execution in our setting.

approach requires only one (custom) program analysis pass instead of several calls to the model checker. Finally, our approach allows for trading accuracy against efficiency with many parameters.

Using maximum satisfiability solving. Jose and Majumdar [131] present a fault localization method based on maximum satisfiability solving. The starting point is an input vector **i** for which the program violates its specification. First, techniques from bounded model checking are used to construct a *trace formula* α, which encodes the semantics of each statement that is executed with input **i**. Each clause of α corresponds to one program statement that has been executed. Second, this trace formula α is extended to an unsatisfiable formula β by additionally asserting that the input variables have the values defined by **i** and the specification *does* hold. Finally, a solver for maximum satisfiability is used to compute a maximum number of clauses of α such that β is satisfiable. The complement of the corresponding set of statements is reported as a diagnosis because these statements can be modified (simultaneously) to render the program correct. Since there can be several different maximum satisfiability solutions, all of them are computed and reported. This approach has been implemented for C programs in the tool Bug-Assist [130]. We used Bug-Assist as a baseline for comparison in our experimental results.

In contrast to this procedure, our model-based diagnosis approach considers several input vectors simultaneously. The results of our experimental evaluation are thus not surprising: our approach reports fewer potential fault locations but is also slower. Another difference is in the computation of diagnoses. Our approach uses a hitting set tree algorithm [182] to derive diagnoses from conflicts, which are essentially unsatisfiable cores. Jose and Majumdar compute diagnoses directly as maximum satisfiability solutions. Interestingly, Fu and Malik [94] point out that the maximum satisfiability problem can be solved efficiently by eliminating unsatisfiable cores. This unsatisfiable core elimination is similar to our procedure for deriving diagnoses from conflicts. Thus, in a sense, our algorithm also computes maximum satisfiability solutions (the diagnoses) but via the elimination of unsatisfiable cores (the conflicts). Other differences concern the program analysis. Our approach uses symbolic or concolic execution and does not only consider one execution paths through the program.

Program slicing. Program slicing, introduced by Mark Weiser [211], is a technique to compute a set of program statements that may influence the values of given program variables at a given point in the program. Program slicing can be used for fault localization by computing statements that may have contributed to a failure (e.g., an assertion violation or a wrong output produced by a program). A survey of existing techniques and applications is given by Tip [207]. Program slicing is mostly based on analyzing data flow and control flow dependencies, and can thus be considered as a rather lightweight technique. In contrast, our model-based approach also considers the semantics of the program statements. It is thus more expensive but potentially also more accurate.

Delta debugging. Delta debugging [213] by Zeller and Hildebrandt is a technique to isolate the trigger of a failure. Given an input sequence that causes a program to fail, it computes a minimized version of the input sequence that still causes the failure. In this sense, delta debugging does not directly compute diagnoses in the form of program parts that may be responsible for a failure. It rather supports the developer in understanding the problem. Our fault localization (and correction) approach is thus orthogonal to delta debugging.

5.4 Program Repair

The problem of automatic program repair has already been tackled from various directions. In the following discussion, we mostly focus on synthesis-based approaches for software, but mutation-based solutions and genetic algorithms will be covered as well.

Program sketching. Our method to synthesize repairs from given diagnoses is very similar to program sketching as introduced by Solar-Lezama [197, 196]. The general principle of program sketching has already been outlined in the introduction: the user writes a program with "holes" in it and provides a specification. A synthesis tool then computes implementations for the holes such that the specification is satisfied for all inputs. In the approach by Solar-Lezama, holes are unknown integer values. For synthesizing more complex program parts, the user has to provide so-called *generators*, which are program fragments containing only integer holes. Loops in the program are unrolled for a fixed number of times. Function calls are inlined (with a user-given bound for recursive functions). The specification can be given using reference implementations and assertions. CEGIS (see Section 2.7) is used as algorithm to synthesize solutions. The approach is implemented in the tool Sketch, which was used as a baseline for comparison in our experimental evaluation (see Section 4.9.3.4). Sketch encodes program correctness in a propositional formula and uses a SAT solver in the CEGIS loop.

Our repair approach is similar in various respects. We also use CEGIS to synthesize repairs, but build on SMT solving (with configurable theories) rather than plain SAT solving. Furthermore, we present heuristics to speed up the convergence of the CEGIS loop. We also show how CEGIS can be interleaved with on-the-fly program analysis to obtain more focused information about the program correctness on demand. Our templates have a similar role as the generators in program sketching but are applied fully automatically. We also combine repair synthesis with fault localization to realize a fully automatic debugging flow. Finally, our means of program analysis can be configured with many parameters to trade accuracy for efficiency. All these ingredients contribute not only to better scalability compared to Sketch (see Section 4.9.3.4) but also to the computation of fine-grained and readable repairs.

Syntax-guided synthesis. Alur et al. [4] propose to define a synthesis problem not only by means of a semantic correctness specification but also by syntactic constraints on the solution. More specifically, a *syntax-guided synthesis* problem is defined to consist of (1) a formula $\varphi(v_a, v_b, \ldots, f_1, f_2, \ldots)$ as a semantic correctness specification for a set of function $f_1, f_2, \ldots$ to be synthesized, where $v_a, v_b, \ldots$ are free variables, (2) a set of grammars defining syntactic sets $L_1, L_2, \ldots$ of candidate implementations, and (3) a background theory. The problem asks to find implementations $e_1 \in L_1, e_2 \in L_2, \ldots$ of the functions $f_1, f_2, \ldots$ such that $\forall v_a, v_b, \ldots \colon \varphi(v_a, v_b, \ldots, e_1, e_2, \ldots)$ is valid in the background theory. Alur et al. also define the SYNTH-LIB format as standardized input format for syntax-guided synthesis. It is based on SMT-LIB2 [13], which is the standard input format for SMT solvers. Since 2014, there is a yearly competition, called SyGuS-COMP[7], among syntax-guided synthesis tools based on this format.

Our software repair synthesis problems can be defined as syntax-guided synthesis problems. A correctness formula $\varphi(\bar{i}, f_{c_1}, f_{c_2}, \ldots)$ can be computed similar to Equation 4.6 but with functions $f_{c_1}, f_{c_2} \ldots$ instead of the templates for the components $c_i \in \Delta$. The templates can be supplied separately as syntactic constraints in the form of grammars. This way, any tool from the SyGuS-COMP can be used to synthesize repairs in our basic approach. Our advanced approach, where repair synthesis is interleaved with on-the-fly program analysis to compute the correctness specification lazily, cannot directly be realized via syntax-guided synthesis problems, though.

Functional synthesis. Kuncak et al. [148] address the problem of synthesizing functional code that satisfies a given input/output relation. Similar to this thesis, the vision is in combining the imperative and the declarative programming paradigm: instead of implementing certain code blocks, the user is allowed to specify their behavior declaratively. A synthesizing compiler then inserts a concrete implementation automatically. Kuncak et al. [148] focus on synthesis procedures for specifications defined as formulas in linear rational arithmetic, linear integer arithmetic, or formulas over sets with size constraints. The approach has been implemented in Comfusy [147], which is a synthesizing compiler for Scala. Functional synthesis procedures have also been proposed for unbounded bitvector arithmetic [200] or algebraic data types and arrays [118]. As a fundamental difference to our work, functional synthesis assumes the specification to be local, so an implementation of the specification can be synthesized without considering other program parts. In contrast, our repair synthesis approach synthesizes program parts to make the entire program satisfy a global specification. This also requires suitable techniques for program analysis.

Path-based program repair. Similar to our work, Riener et al. [185] also propose a repair approach based on iterative refinements. The specification is given using pre- and postconditions. The fault location is assumed to be known. In every iteration, a model checker is used to compute a symbolic

[7] http://www.sygus.org/ (last visit on 2015-08-01).

counterexample, which keeps the data flow symbolic but the control flow concrete. Based on this counterexample, a verification condition is then computed at the point of the faulty program part. Intuitively, this is done by pushing the precondition down and the postcondition up along the counterexample path using Hoare axioms (see Section 2.8.4). This transforms the global specification into a local one (for one counterexample path). The final step of the loop is to resynthesize the faulty program part such that all symbolic counterexamples seen so far are resolved. This is done by interpreting the verification conditions computed so far as synthesis goal and applying functional synthesis, as discussed in the previous paragraph. If the functional synthesis procedure reports unrealizability, the loop aborts. Otherwise, a model checker verifies the candidate. If it finds a counterexample, another iteration is started.

There are several differences to our work. The approach by Riener et al. does not use templates. Depending on the underlying synthesis procedure, this can result in unreadable repairs, which makes it difficult to keep the user in the loop. In particular, the proof-of-concept implementation for C programs applies bit-blasting, computes the repair as a network of AND-gates, and maps this network back to C code. On the other hand, the approach is flexible regarding the synthesis procedure and a realization using templates is possible as well. Moreover, avoiding templates avoids the disadvantage that no repair may be found even if one exists. The principle of considering more and more execution paths is similar to our repair approach with on-the-fly program analysis, but we consider more and more input vectors, which indirectly results in analyzing more and more execution paths. Our approach computes the effect of the behavior of components to repair on the global specification rather than breaking down the global specification into local constraints for the component to repair. Finally, the approach by Riener et al. does not handle cases where the faulty program part is executed multiple times along one counterexample path (e.g., because it occurs in a loop or in a function that is called multiple times). The simultaneous computation of repairs for multiple program parts is not addressed either. Our repair approach can handle these cases seamlessly.

Repair as a game. Jobstman et al. [127, 128] consider the repair of finite-state programs with respect to a given LTL specification. The problem is transformed into a finite-state game and a repair is computed as a strategy in this game. Fault localization is done simultaneously by having the strategy select a component to repair as a first move. To support the readability of repairs, heuristics attempt to compute a memoryless strategy that does not introduce additional state variables. Still, the repair can be an arbitrary function over state variables and inputs, and thus quite difficult to understand. In contrast, our template-based approach restricts the shape of solutions, so the result will always be readable.

Using predicate abstraction. Griesmayer et al. [104] extend the idea of computing a repair as a strategy in a game to software with a potentially infinite state space using predicate abstraction. A repair is computed for a Boolean abstraction of the program (as computed by verification tools like SLAM [12]). The abstraction is such that the correctness of the abstract program implies the correctness of the original program. A repair for the abstract program thus suggests a repair of the concrete one. However, if no repair of the abstract program exists, this does not mean that no repair of the concrete program exists. The approach by Griesmayer et al. does not include a refinement of the abstraction in this case and is thus correct but incomplete. Our approach of incomplete program analysis (considering only a subset of the execution paths) can also be seen as abstraction. Our repair approach with on-the-fly program analysis can even be understood as counterexample-guided abstraction refinement.

Mutation-based repair. A mutation is a small syntactic modification (like changing a "+" into a "−") of the program (see also Section 4.2.1). Debroy and Wong [74] propose a simple software repair method based on such mutations. First, statistical methods for fault localization are used to rank statements according to their likelihood of containing faults. Second, starting with the statements ranked highest, mutations are introduced. Each mutated program is checked for correctness until a repair is found. A similar approach is used by Raik et al. [179] for hardware designs at the register-transfer level. FoREnSiC [30] also contains a back end implementing this approach in combination with program slicing [183]. While this method can scale up to very large programs, it is rather restricted in its fault model. In contrast, our fault model of incorrect expressions can also handle faults that can only be fixed with more substantial changes in the code (e.g, changing some expression "`a+2`" to "`b-823`", which may be too far-fetched for a mutation).

Genetic approaches. Arcuri [8] uses genetic programming algorithms to evolves the incorrect program with guidance from a fitness function that counts how many test cases pass. Each iteration keeps a population of programs. The fitness function selects promising candidates. Crossover and mutation operators are then used to generate "offspring", which is the population of the next iteration. Arcuri and Yao [10, 9] extend this approach to a setting where both the program and the test cases co-evolve competitively. Le Goues et al. [102] present GenProg, which is a similar approach to repair an incorrect program using genetic programming and test cases as a specification. It uses structural differencing [3] and delta debugging [213] to reduce the difference to the original program. Furthermore, it focuses the genetic operators to statements that are executed in failing test cases.

From an algorithmic point of view, our program repair approach is fundamentally different from genetic approaches. Nevertheless, there are similarities in the iterative refinement of candidates. While genetic approaches have the potential to scale up to large programs (GenProg [102] has been used on benchmarks with several thousand lines of code), our constraint-based approach is more systematic.

Concurrent programs. Vechev et al. [210] present an approach to repair concurrent software programs by synthesizing atomic sections (where no context switch is allowed) that are sufficient to guarantee correctness. Similar to the work by Griesmayer et al. [104] (discussed above), the approach is also based on abstraction. However, the abstraction can also be refined based on counterexamples. If a counterexample is found, heuristics are used to decide whether to refine the abstraction or to insert an atomic section. The user has to provide a characterization of bad states (e.g., using assertions) as a specification. In previous work [35] that is orthogonal to this thesis, we present an alternative solution that uses abstraction by means of uninterpreted functions. The behavior of sequential executions is taken as implicit specification, so the user does not have to (but can) provide an explicit specification. Cerný et al. [53, 54] present an approach where counterexamples cannot only be eliminated by adding synchronization but also by instruction reordering. Khoshnood et al. [135] present ConcBugAssist, an approach that diagnoses concurrency bugs using maximum satisfiability solving (similar to Bug-Assist [131, 130]; see above) and computes repair suggestions by solving a binate covering problem. The repair process yields inter-thread ordering constraints that are then enforced using synchronization primitives such as locks, signal/wait, or atomicity primitives.

These works assume that the program is correct if executed sequentially, i.e., the incorrectness is only in missing synchronization. Since the program will always be correct if all threads are made atomic, this can actually be seen as an optimization problem to find a minimum set of atomic sections that suffices. In contrast, this thesis focuses on the repair of sequential programs an is thus orthogonal.

6 Conclusion and Outlook

In this thesis, we proposed novel satisfiability-based methods for controller synthesis, both for hardware systems and for software programs. In the hardware setting, we focused on scalability for safety specifications. In the software setting, we concentrated on controller synthesis techniques in the application of automatic program repair. In order to wrap this thesis up, the following sections will now recapitulate our motivation and contributions, discuss the main conclusions, and give suggestions for future work.

6.1 Summary

Before coming to conclusions, a summary of the motivation and contributions of this thesis is appropriate.

Synthesis versus verification. In the conventional development process for obtaining correct hardware or software programs, the developer (1) implements the design intent manually, (2) verifies the implementation, e.g, by executing test cases or by applying formal verification techniques such as model checking, and (3) fixes all the bugs that are discovered in the verification phase. Synthesis can simplify this process significantly by automatically computing a correct implementation from a specification that formalizes the design intent declaratively. Writing such a specification can be significantly easier than implementing it because the specification only expresses *what* the system shall do, but not *how*.

Challenges in synthesis. The advantages of synthesis also come with several challenges. First, there is a scalability challenge induced by the high worst-case complexities. To achieve acceptable scalability in practice, symbolic algorithms are important. While Binary Decision Diagrams (BDDs) are the predominant technology in realizing synthesis algorithms symbolically, there has been enormous progress on decision procedures for the satisfiability of formulas in the recent years and decades. Yet, their potential in synthesizing hardware components is still largely unexplored. A second challenge lies in the applicability of synthesis. Formalizing the design intent completely in a declarative fashion can be unmanageable in practice. A more promising direction is controller synthesis, where parts of the synthesis problem can be defined imperatively and other parts declaratively. This gives rise to interesting applications such as program sketching and automatic program repair. These applications have not yet received the research attention they deserve either. This thesis focused on these two challenges. Other synthesis challenges that are mostly orthogonal to this thesis include the quality of synthesized systems, techniques for specification engineering and debugging, and appropriate specification languages.

Contributions to the scalability challenge. To address the scalability challenge, we proposed several novel algorithms to synthesize hardware controllers from safety specifications. Our algorithms make use of various techniques such as query learning, templates to fix the structure of solutions, interpolation, and satisfiability certification. They are tailored towards decision procedures for the satisfiability of propositional formulas (SAT solvers), Quantified Boolean Formulas (QBF solvers), or solvers for Effectively Propositional Logic (EPR), exploiting solver features such as incremental solving or the computation of unsatisfiable cores where possible. We also presented numerous optimizations, including heuristics to partially expand quantifiers, optimizations using information about unreachable states or variable independencies, as well as low-level optimizations in the formula encoding. In an experimental evaluation, we compared our methods and optimizations on a broad range of benchmarks. The comparison also included a BDD-based tool, a highly optimized state-of-the-art synthesis tool, and an alternative SAT-based approach as a baseline. Our implementation is available with the open-source tool Demiurge, which already won two gold medals in an international synthesis competition.

Contributions to the applicability challenge. To address the applicability challenge, we presented an approach for controller synthesis in the application of automatic software program repair, using assertions in the code as specification. Our specific goal was to synthesize fine-grained and readable repairs. To achieve this goal, we proceeded in three steps. First, we used symbolic or concolic execution in a program analysis step to lift the repair problem into the domain of logic. Second, we proposed a fault localization approach based on the fault model of incorrect expressions and techniques from model-based diagnosis with Satisfiability Modulo Theories (SMT) solving. We also discussed an alternative for fault localization using deductive verification and pre- and postconditions as specification. The (potential) fault locations reported in the second step were then used as a basis for synthesizing replacements in the last step. Similar to our algorithms for hardware synthesis, we built on templates to fix the structure of the solution and counterexample-guided refinement of solution candidates to find correct repairs. We discussed some practical issues with this approach and addressed them in two ways. First, we developed heuristics to speed up the repair candidate refinement. Second, we proposed an improved flow that interleaves the repair synthesis with on-the-fly program analysis to obtain more focused information about the program behavior. We prototyped our approach for simple C programs in the open-source tools FoREnSiC and Frama-C. Our approach as well as the implementation is highly configurable with many parameters to trade accuracy for efficiency. Finally, we presented experimental results evaluating the usefulness of the produced diagnostic information and the performance in various configurations.

6.2 Conclusions

Chapters 3 and 4 already discussed the strengths and weaknesses of the different algorithms and optimizations while they were presented. Moreover, the Sections 3.3.5 and 4.9.4 summarized the most important conclusions that can be drawn from our experiments. In this section, we will not repeat this discussion but rather focus on the most important conclusions from a high-level point of view. This will also form the basis to our suggestions for future work. While the main chapters were organized with respect to the setting (hardware or software), we will organize the conclusions according to the challenges (scalability and applicability) addressed by this thesis.

6.2.1 Scalability

The scalability results achieved in this thesis rely on a multitude of factors.

Exploiting solver features. In contrast to verification, decision procedures that can only give a yes/no answer are of no use in synthesis. Fortunately, many decision procedures for satisfiability are based on the search for satisfying structures. These artifacts can in turn be used to build an implementation for a given specification in synthesis. Modern SAT-, QBF- and SMT solvers offer additional features that can be exploited in synthesis as well. This includes the computation of unsatisfiable cores, which can be used to generalize discovered facts. Another example is incremental solving, which can be used to answer sequences of similar queries much more efficiently. Our algorithms for hardware controller synthesis utilize such solver features by design, which turned out to be crucial for being competitive with BDDs. Our repair approach for software programs also achieves speedups of up to two orders of magnitude for individual debugging steps by exploiting these solver features.

Counterexample-guided refinement. The general algorithmic principle of refining solution candidates iteratively based on counterexamples turned out to be a good match with decision procedures for the satisfiability of formulas. We used this concept in two flavors: query learning and Counterexample-Guided Inductive Synthesis (CEGIS). Query learning combined with SAT solving proved to be our best approach in our hardware controller synthesis experiments. This applies both to the first step of computing a winning strategy as well as to the second step of constructing a circuit. In the second step, query learning also produced circuits that were smaller by more than one order of magnitude on average compared to other techniques such as interpolation, QBF certification, or the BDD-based cofactor approach. This suggests that query learning performs well at exploiting available implementation freedom. In the software setting, CEGIS turned out to be efficient, but we could still speed up its convergence with heuristics. Furthermore, CEGIS was amenable for our improved repair synthesis approach that interleaves repair candidate refinements with on-the-fly program analysis.

Abstracting program semantics. In the domain of software controller synthesis, our approach for program repair provides many parameters to trade accuracy for efficiency. This turned out to be very important for handling larger programs. In the TCAS case study, our approach could synthesize repairs in reasonable time by abstracting program variables with mathematical integers and using an SMT solver. In contrast, the existing tool Sketch could only solve manually simplified problems because of using bit-precise reasoning and complete program analysis.

Handling quantifiers. The game-based approach to synthesis inherently involves dealing with both universal and existential quantifiers. The support for both quantifiers is also among the reasons for the sustained success of BDDs in hardware synthesis. When switching from BDDs to decision procedures for satisfiability, one could thus expect that QBF solvers are the most suitable choice. Yet, in our experiments, our algorithms using plain SAT solving outperformed the QBF-based algorithms significantly, even though (often far) more solver calls are necessary to compensate for the lack of universal quantifiers. Our heuristic for quantifier expansion reduces this amount of iterations at the cost of larger formulas for the SAT solver, which gives a speedup of one more order of magnitude. This suggests that the current state in QBF solving is still lacking behind its potential, at least for the specific kinds of QBF problems we encounter in our synthesis algorithms. However, considering that QBF is still a rather young research discipline compared to SAT, this situation may change in the future.

More expressive logics. The scalability of our approach based on reduction to EPR, which is a more expressive logic, is even worse than when using QBF in our experiments. Together with the statement from the previous paragraph, this suggests that breaking the synthesis problem into simple solver queries in a lean logic is a better strategy than delegating bigger chunks of the problem to the underlying solver.

Parallelizability. Since our satisfiability-based methods for hardware controller synthesis mostly break the synthesis problem down to many small solver queries that do not crucially depend on each other, they are also well suited for fine-grained application-level parallelization. This stands in contrast to symbolic algorithms realized with BDDs, which are often intrinsically hard to parallelize [84]. In this thesis, we presented parallelizations that do not only exploit hardware parallelism but also combine different (variants of) algorithms in different threads. This way, we achieved average speedups of around one order of magnitude with only three threads.

Outperforming BDDs. Due to our heuristics and optimizations, careful utilization of solver features, and our parallelization, our satisfiability-based methods managed to outperform a BDD-based synthesis tool by more than one order of magnitude regarding execution time, and even two orders of magnitude regarding circuit size on average in our experiments. Our parallelization is even competitive with AbsSynthe, a highly optimized state-of-the-art tool implementing advanced optimizations such as abstraction/refinement. These results confirm that decision procedures for the satisfiability of formulas can indeed be used to build scalable synthesis algorithms, as claimed in the thesis statement in Section 1.3.

There is no silver bullet. Despite the excellent performance results we achieved on the average in our hardware synthesis experiments, we also observed that different techniques perform well on different classes of benchmarks. Our main contribution regarding scalability can thus be seen in extending the portfolio of available synthesis approaches with new algorithms that complement existing techniques.

Safety specifications. Our hardware controller synthesis algorithms operate on safety specifications. Many of the benchmarks used in our experimental evaluation originally contained liveness properties that have been translated to safety specifications by imposing fixed bounds on the reaction time. While choosing low bounds for the reaction time (such that the specification is still realizable) can have the advantage of producing systems that react faster, the translation may have a negative performance impact compared to handling liveness properties directly in the synthesis algorithm.

6.2.2 Applicability

Besides focusing on controller synthesis, which allows for a mixed imperative/declarative programming paradigm, we address the applicability challenge mostly through our approach for applying controller synthesis in the context of automatic program repair. This approach is appealing in various respects.

Fine-grained repairs. Our choice of incorrect expressions as a fault model results in fine-grained repairs, i.e., repairs that differ from the original program only in small parts. Keeping as much of the original code as possible increases the chances that the synthesized repair coincides with the design intent by the user. It also supports the understandability of computed repairs because the user only needs to comprehend the consequences of small code changes (rather than, e.g., completely resynthesized functions). Finally, it also supports performance because the restriction to resynthesizing only small code parts reduces the search space for repairs.

Human-readable repairs. Our template-based approach for synthesizing repairs allows us to control the shape of the computed repairs. We start with simple templates and switch to more expressive ones upon failure. This reduces the search space for repairs and thus supports scalability. Moreover, it ensures that the computed repairs are human readable, which is important for keeping the user in the loop.

Keeping the user in the loop. In the envisioned usage scenario, the tool can suggests repairs, but it is always the user who selects one. After all, it is also the user who must continue to work with the repaired program (extend it with new features, refactor it, fix other bugs, etc.). Another flavor of keeping the user in the loop regards the debugging process itself. In our approach, we do not assume the specification to be complete in the sense that it rules out every behavior that conflicts with the design intent. If only undesirable repairs are presented by the tool, the user can refine the specification and start the tool again. Readability and fine-grainedness of computed repairs are important usability features to enable this.

Specification mechanisms. We support various kinds of specifications. Natively, our approach uses assertions in the code. Such assertions can also be used to compare results with that of a reference implementation. Our improved repair approach with on-the-fly program analysis also supports test cases. Our fault localization variant based on deductive verification uses pre- and postconditions as well as user-defined loop invariants. This has the advantage that functions in the code can be processed in isolation, which potentially increases the scalability. The disadvantage is that writing function contracts and appropriate loop invariants can be a lot of manual work.

Generality. Our fault model and our template-based approach for restricting the shape of repairs both have the positive effect of narrowing down the search space and producing useful repairs if successful. Yet, they may also prevent our approach from finding a repair. While we have encountered such cases in our experiments, we have also encountered other cases where repairs are found even though the fault model does not match. We thus conclude that our approach forms a reasonable compromise between generality on one hand and scalability and usefulness of the results on the other hand.

Configuration. Our repair approach provides many configuration parameters to trade accuracy for efficiency. This can be both a blessing and a curse. On the one hand, our approach can be tailored towards a broad range of problems. One the other hand, the experience with our tool shows that it often takes some attempts before a suitable parameter configuration is found.

Potential. In support of the second claim of the thesis statement (see Section 1.3), we have demonstrated that satisfiability-based methods for controller synthesis can be combined seamlessly with program analysis and fault localization techniques to realize a fully automatic, yet flexible flow for software program repair. We believe that this application is particularly promising for making synthesis techniques accessible to a broader audience of potential users. But program repair is of course not the only interesting application of synthesis. In related work, we have also worked on runtime enforcement for reactive systems [37], program sketching for concurrent systems [28], and synthesis of synchronization for concurrent software programs [35].

6.3 Future Work

There is a multitude of directions for future work on improving our satisfiability-based hardware controller synthesis methods as well as our application of controller synthesis in the field of program repair.

6.3.1 SAT-Based Hardware Controller Synthesis

Our suggestions for future work in satisfiability-based hardware controller synthesis range from improvements in the underlying reasoning engines up to extensions for different classes of specifications.

QBF preprocessing. While our QBF-based algorithms for hardware controller synthesis were not among the best solutions in our experiments, we still observed that using incremental QBF solving and QBF preprocessing both can have a very positive performance impact. Researching ways to combine these techniques therefore seems to be a particularly promising direction to support the success of QBF in synthesis. Furthermore, in our circuit computation method based on QBF certification, preprocessing could not be applied because existing tools only preserve satisfying assignments for existentially quantified variables [189], but are in general not certificate preserving. Research on such certificate-preserving preprocessing solutions could thus boost the performance of QBF certification (not only) in synthesis.

Solver parameters. So far, we used all solvers with default parameters in our experiments. It is not unlikely that a solid speedup can be achieved by tuning solver parameters to the specific kinds of decision problems that are encountered in our synthesis algorithms. For instance, our algorithms based on SAT solving usually make huge amounts of rather simple queries. The default parameters of the SAT solvers may be tuned to more complex instances from SAT competitions, however.

Other logics. Our approach based on reduction to EPR did not perform well in our experiments. For this reason, we did not explore the alternative of using DQBF instead. Yet, recent progress [92, 93, 11, 96] in theory and tools for DQBF makes this approach interesting as well.

Computing multiple interpolants. Some of our methods to compute circuits from given strategies are based on interpolation. As mentioned in Section 5.1, it would be interesting to also implement the approach by Hofferek et al. [113] for computing multiple interpolants from a single proof.

Reachability optimization. Our reachability optimization is rather simplistic and still has a very positive performance impact. Other variants may thus yield even bigger speedups. In particular, our reachability optimization avoids the explicit computation of an over-approximation of the reachable states. Exploring this option based on existing work in verification [161] can be worthwhile.

Parallelization. Our parallelized hardware synthesis method merely demonstrates that a parallelization is easily possible and beneficial for our satisfiability-based synthesis methods. However, it is in no way optimal. First, there is a plethora of possibilities to combine different algorithms, optimizations and solver configurations in different threads. Second, there are also numerous ways for exchanging information between the threads. A thorough exploration of possibilities is still to be done.

AIGER as symbolic data structure. Another alternative to BDDs is to use AIGER circuits as a data structure for Boolean formulas. The standard Boolean connectives ($\wedge, \vee, \rightarrow, \ldots$) are easy to realize by adding gates accordingly. Universal and existential quantification can be realized by expansion. Circuit simplification techniques as implemented in ABC [42] can be applied to reduce the size of the symbolic representation after applying operations (similar to variable reordering in BDDs). A SAT solver can be used for equivalence or inclusion checks. In contrast to BDDs, such a symbolic representation is not canonical. It may thus be more compact in cases where BDDs explode in size (see Section 2.3.1).

Specification preprocessing. Inspired by the formidable performance impact of preprocessing in QBF solving, research on preprocessing techniques for specifications in synthesis can be another angle from which the scalability issue can be tackled. For specifications defined as AIGER circuits, one first idea would be to develop heuristics for identifying auxiliary variables (outputs of AND-inverter gates defining the transition relation) that can be controlled fully and independently by either the system or the environment. As a simple example, some auxiliary variable t may be defined as a function over some vector $\overline{i}_t \subseteq \overline{i}$ of uncontrollable inputs, and the inputs $\overline{i}_t$ are used nowhere else. Such auxiliary variables can be replaced by new controllable or uncontrollable inputs, and their respective cone of influence can be removed. Another idea is to detect monotonic dependencies of the error output on inputs or latches and replacing such inputs or latches with constants. Existing techniques for circuit simplification can also be applied, of course.

Other specifications. Our satisfiability-based hardware controller synthesis algorithms operate on safety specifications. A natural point for future work is thus to extend them to other types of specifications. Interesting cases would include reachability specifications (some states must be visited at least once), Büchi specifications (some states must be visited infinitely often), or even GR(1) [36] (see also Section 2.5.4). Our methods to compute a circuit from a given strategy are rather agnostic against the specification from which the strategy has been constructed. Here, future work would mostly be in working out an efficient implementation. For the computation of strategies, the situation is different though. Learning-based algorithms are not difficult to define for other specification formats in principle. If and how they can be applied *efficiently* remains to be explored, though.

6.3.2 SAT-Based Software Controller Synthesis for Program Repair

All steps of our software repair approach, from program analysis to fault localization and repair synthesis, can still be improved by future work.

Automatic specifications. Certain correctness properties of a program can be defined fully automatically. For instance, if the program performs a division, the divisor must not be zero. If the program accesses an array element, the index must not exceed the array size. In our current realization, such properties need to be defined manually using assertions. Automating such checks, either by automatically inserting assertions into the code or by including these checks directly in the program analysis, would improve the usability.

Fault model. As already pointed out in Section 4.2.1, our fault model of incorrect expressions can be extended in various ways, but at the cost of an increased computational effort in automatic debugging. It would thus be interesting to explore extensions to the fault model and study the different trade-offs that can be achieved between generality and efficiency.

Memory model. Our current implementation of the program analysis step using concolic execution cannot reason about array indices or pointer values symbolically, but applies abstraction by taking the concrete index or pointer value for memory accesses. Consequently, our tool cannot repair bugs in array index computations or pointer arithmetic. This shortcoming can be mitigated by extending our implementation with a memory model and support for the theory of arrays in SMT solving.

Using KLEE [51]. Our concolic execution engine is based on CREST [50] and is thus rather robust and highly configurable. Our symbolic execution engine, on the other hand, is merely a self-made proof-of-concept implementation. By extending a mature symbolic execution engine such as KLEE [51] so that it can compute diagnostic information as needed for our fault localization and correction approach, it is possible to realize a more attractive alternative to concolic execution in FoREnSiC.

Using Bounded Model Checking (BMC). Another interesting extension would be to use BMC tools such as CBMC [61] as an alternative to symbolic or concolic execution for program analysis. Both techniques have their strengths and weaknesses.

Ranking diagnoses. Our fault localization approach based on Model-Based Diagnosis (MBD) has the advantage of being very systematic, but the disadvantage of producing many diagnoses in some cases. On the other hand, statistical techniques such as spectrum-based fault localization [2] can rank program parts regarding their likelihood of being responsible for the incorrectness. These two techniques can also be combined by ranking the diagnoses produced by MBD. The repair engine can then repair the most likely diagnoses first, and may thus find a fix faster.

Reducing undesirable repairs. The debugging experiments on the DLX processor design (see Section 4.9.2.4) revealed cases where faults are repaired by making the program terminate before critical assertions can be reached. In most cases, such repairs are undesirable. Future extensions of FoREnSiC could eliminate such undesirable repairs by requiring not only correctness in the sense that assertion violations are impossible, but also that code that was reachable in the original program does not become unreachable by the repair.

Deductive program repair. Our alternative debugging approach based on deductive verification and pre- and postconditions as a specification has only been worked out for fault localization in this thesis. However, since diagnoses are computed by checking the program for repairability, it is conceptually simple to extend this approach also to program repair: if the underlying theorem prover is able to deliver a witness for the validity of the repairability formula (in the form of Skolem functions), then this witness directly describes a repair[1]. It would be interesting to see if repairs computed as Skolem functions are readable enough to constitute a viable alternative for our template-based approach.

Syntax-guided synthesis. In our discussion of related work (Section 5.4), we have already pointed out that our repair synthesis problems can be expressed as syntax-guided synthesis problems as defined by Alur et al. [4]. By implementing a translator into the standardized input format for syntax-guided synthesis, any tool from the SyGuS-COMP[2] competition could be plugged into our automatic software debugging flow (on-the-fly program analysis cannot be supported in this mode, though).

[1] Given that the variables in the formula can be linked back to the corresponding program variables.

[2] `http://www.sygus.org/` (last visit on 2015-08-01).

Bibliography

[1] Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal*, 23(2):297–322, 2015. (Cited on page 156.)

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009. (Cited on page 166.)

[3] Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffX: An algorithm to detect changes in multi-version XML documents. In *Centre for Advanced Studies on Collaborative Research (CASCON'05)*, pages 1–11. IBM, 2005. (Cited on page 160.)

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD'13)*, pages 1–8. IEEE, 2013. (Cited on pages 158 and 167.)

[5] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *STTT*, 7(2):118–128, 2005. (Cited on page 153.)

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008. (Cited on page 38.)

[7] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987. (Cited on pages 6 and 30.)

[8] Andrea Arcuri. On the automation of fixing software bugs. In *International Conference on Software Engineering (ICSE'08)*, pages 1003–1006. ACM, 2008. (Cited on page 160.)

[9] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011. (Cited on page 160.)

[10] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation (CEC'08)*, pages 162–168. IEEE, 2008. (Cited on page 160.)

[11] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 523:86–100, 2014. (Cited on page 165.)

[12] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001. (Cited on pages 1 and 159.)

[13] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories (SMT'10)*, 2010. (Cited on pages 17, 22 and 158.)

[14] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [25], pages 825–885. (Cited on pages 6, 17 and 22.)

[15] Bernd Becker, Rüdiger Ehlers, Matthew D. T. Lewis, and Paolo Marin. ALLQBF solving by computational learning. In *Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2012. (Cited on page 154.)

[16] Marco Benedetti and Hratch Mangassarian. QBF-based formal verification: Experience and perspectives. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 5(1-4):133–191, 2008. (Cited on page 21.)

[17] Dietmar Berwanger, Krishnendu Chatterjee, Martin De Wulf, Laurent Doyen, and Thomas A. Henzinger. Alpaga: A tool for solving parity games with imperfect information. In *Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS'09), volume 5505 of *Lecture Notes in Computer Science*, pages 58–61. Springer, 2009. (Cited on page 155.)

[18] Dietmar Berwanger, Krishnendu Chatterjee, Martin De Wulf, Laurent Doyen, and Thomas A. Henzinger. Strategy construction for parity games with imperfect information. *Information and Computation*, 208(10):1206–1220, 2010. (Cited on page 155.)

[19] Dirk Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015. (Cited on page 125.)

[20] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007. (Cited on pages 1 and 125.)

[21] Armin Biere. Resolve and expand. In *Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2004. (Cited on pages 20 and 21.)

[22] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008. (Cited on page 83.)

[23] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT competition 2014. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2 of *Series of Publications B*, pages 39–40. Department of Computer Science, University of Helsinki, 2014. (Cited on pages 83 and 94.)

[24] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. (Cited on page 1.)

[25] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. (Cited on pages 6, 19, 169, 174, 177 and 179.)

[26] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Conference on Automated Deduction (CADE-23)*, volume 6803 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2011. (Cited on pages 20, 21, 46 and 83.)

[27] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Informatica*, 51(3-4):193–220, 2014. (Cited on page 11.)

[28] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, volume 9035 of *Lecture Notes in Computer Science*, pages 517–532. Springer, 2015. (Cited on pages 10 and 164.)

[29] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - A new requirements analysis tool with synthesis. In *Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010. (Cited on pages 3, 10, 11, 89 and 155.)

[30] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. FoREnSiC - An automatic debugging environment for C programs. In *Haifa Verification Conference (HVC'12)*, volume 7857 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2012. (Cited on pages 9, 13, 103, 131, 153 and 159.)

[31] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. SAT-based methods for circuit synthesis. In *Formal Methods in Computer-Aided Design (FMCAD'14)*, pages 31–34. IEEE, 2014. (Cited on pages 9, 10, 13, 41 and 153.)

[32] Roderick Bloem, Rüdiger Ehlers, Swen Jacobs, and Robert Könighofer. How to handle assumptions in synthesis. In *Workshop on Synthesis (SYNT'14)*, volume 157 of *EPTCS*, pages 34–50, 2014. (Cited on page 11.)

[33] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007. (Cited on pages 84 and 85.)

[34] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In *Workshop on Synthesis (SYNT'12)*, volume 84 of *EPTCS*, pages 47–53, 2012. (Cited on page 10.)

[35] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design (FMCAD'14)*, pages 35–42. IEEE, 2014. (Cited on pages 10, 160 and 164.)

[36] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012. (Cited on pages 3, 28, 29, 155 and 166.)

[37] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: Runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, volume 9035 of *Lecture Notes in Computer Science*, pages 533–548. Springer, 2015. (Cited on pages 10 and 164.)

[38] Roderick Bloem, Robert Könighofer, Franz Rock, and Michael Tautschnig. Automating test-suite augmentation. In *Quality Software (QSIC'14)*, pages 67–72. IEEE, 2014. (Cited on page 11.)

[39] Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*, volume 8318 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014. (Cited on pages 8, 9, 10, 13, 41 and 153.)

[40] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In *Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012. (Cited on pages 85 and 155.)

[41] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. (Cited on pages 1, 8, 57, 86, 94 and 153.)

[42] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010. (Cited on pages 55, 68, 72, 78, 83, 93, 100 and 165.)

[43] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. AbsSynthe: Abstract synthesis from succinct safety specifications. In *Workshop on Synthesis (SYNT'14)*, volume 157 of *EPTCS*, pages 100–116, 2014. (Cited on pages 7, 86, 93, 94 and 155.)

[44] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009. (Cited on page 22.)

[45] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. (Cited on pages 1, 17 and 18.)

[46] Nader H. Bshouty. Exact learning boolean function via the monotone theory. *Information and Computation*, 123(1):146–153, 1995. (Cited on page 31.)

[47] Uwe Bubeck and Hans Kleine Büning. Bounded universal expansion for preprocessing QBF. In *Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2007. (Cited on page 15.)

[48] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. (Cited on page 3.)

[49] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Logic in Computer Science (LICS'90)*, pages 428–439. IEEE, 1990. (Cited on page 1.)

[50] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering (ASE'08)*, pages 443–446. IEEE, 2008. (Cited on pages 36, 129, 132 and 166.)

[51] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI'08)*, pages 209–224. USENIX Association, 2008. (Cited on pages 1, 35 and 166.)

[52] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), 2008. (Cited on page 35.)

[53] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *Computer Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 951–967. Springer, 2013. (Cited on page 160.)

[54] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. In *Computer Aided Verification (CAV'14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 568–584. Springer, 2014. (Cited on page 160.)

[55] Krishnendu Chatterjee and Thomas A. Henzinger. Assume-guarantee synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2007. (Cited on page 10.)

[56] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks (DSN'02)*, pages 595–604. IEEE, 2002. (Cited on page 156.)

[57] Alonzo Church. Logic, arithmetic, and automata. In *International Congress of Mathematicians (Stockholm, 1962)*, pages 23–35. Institute Mittag-Leffler, Djursholm, 1963. (Cited on page 3.)

[58] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. (Cited on page 155.)

[59] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. (Cited on page 1.)

[60] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. (Cited on page 1.)

[61] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. (Cited on pages 1, 125, 130, 136, 145, 147 and 166.)

[62] 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'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005. (Cited on pages 1 and 125.)

[63] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976. (Cited on pages 1, 7 and 34.)

[64] James S. Collofello and Scott N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989. (Cited on page 5.)

[65] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1501. Morgan Kaufmann, 1993. (Cited on page 156.)

[66] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design (ICCAD'90)*, pages 126–129. IEEE, 1990. (Cited on page 28.)

[67] 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. (Cited on pages 1, 6 and 14.)

[68] Yves Crama and Peter L. Hammer. *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. (Cited on pages 30 and 31.)

[69] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In *Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012. (Cited on pages 8, 38, 115 and 131.)

[70] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with AMPLE. In *Workshop on Automated Debugging (AADEBUG'05)*, pages 99–104. ACM, 2005. (Cited on page 156.)

[71] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI'02)*, pages 57–68. ACM, 2002. (Cited on page 106.)

[72] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. (Cited on pages 38 and 156.)

[73] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (Cited on page 132.)

[74] Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60, 2014. (Cited on pages 106 and 159.)

[75] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968. (Cited on page 36.)

[76] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010. (Cited on page 20.)

[77] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006. (Cited on page 132.)

[78] Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In *Conference on Artificial Intelligence (AAAI'15)*, pages 3738–3745. AAAI Press, 2015. (Cited on page 153.)

[79] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD'11)*, pages 125–134. IEEE, 2011. (Cited on page 1.)

[80] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. (Cited on page 83.)

[81] Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012. (Cited on pages 24, 85 and 155.)

[82] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In *Formal Methods in Computer-Aided Design (FMCAD'12)*, pages 91–100. IEEE, 2012. (Cited on pages 9, 13, 28, 31, 41, 70, 74, 79, 102 and 153.)

[83] E. Allen Emerson and Kedar S. Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(4):527–550, 2003. (Cited on page 154.)

[84] Jonathan Ezekiel, Gerald Lüttgen, and Radu Siminiceanu. To parallelize or to optimize? *Journal of Logic and Computation*, 21(1):85–120, 2011. (Cited on page 163.)

[85] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004. (Cited on page 156.)

[86] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011. (Cited on pages 24, 85 and 155.)

[87] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013. (Cited on pages 3, 10, 154 and 155.)

[88] Robert Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967. (Cited on page 36.)

[89] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In *Computer Aided Verification (CAV'08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 5–10. Springer, 2008. (Cited on pages 1 and 4.)

[90] Gerhard Friedrich and Kostyantyn M. Shchekotykhin. A general diagnosis method for ontologies. In *International Semantic Web Conference (ISWC'05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2005. (Cited on page 156.)

[91] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999. (Cited on page 156.)

[92] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. In *Workshop on Pragmatics of SAT (POS'12)*, 2012. (Cited on pages 65 and 165.)

[93] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. iDQ: Instantiation-based DQBF solving. In *Workshop on Pragmatics of SAT (POS'14)*, volume 27 of *EPiC Series*, pages 103–116. EasyChair, 2014. (Cited on pages 65 and 165.)

[94] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. (Cited on pages 122 and 157.)

[95] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007. (Cited on page 35.)

[96] Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker. Solving DQBF through quantifier elimination. In *Design, Automation & Test in Europe (DATE'15)*, pages 1617–1622. ACM, 2015. (Cited on page 165.)

[97] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. Reasoning with quantified boolean formulas. In Biere et al. [25], pages 761–780. (Cited on page 6.)

[98] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *International Joint Conference on Automated Reasoning (IJCAR'01)*, volume 2083 of *Lecture Notes in Computer Science*, pages 364–369. Springer, 2001. (Cited on pages 21 and 83.)

[99] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005. (Cited on pages 2, 7 and 36.)

[100] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. (Cited on pages 2, 36 and 130.)

[101] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013. (Cited on page 5.)

[102] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. (Cited on page 160.)

[103] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989. (Cited on page 113.)

[104] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to C. In *Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 358–371. Springer, 2006. (Cited on pages 159 and 160.)

[105] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science*, 174(4):95–111, 2007. (Cited on page 156.)

[106] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault localization using a model checker. *Software Testing, Verification & Reliability*, 20(2):149–173, 2010. (Cited on pages 106 and 156.)

[107] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008. (Cited on page 1.)

[108] Leon Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods: Proceedings of the Symposium on Foundations of Mathematics, Warsaw, 2-9 September 1959*, pages 167–183. Pergamon Press, 1961. (Cited on page 65.)

[109] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 36.)

[110] Georg Hofferek. *Controller Synthesis with Uninterpreted Functions*. PhD thesis, Graz University of Technology, July 2014. (Cited on page 154.)

[111] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *Formal Methods and Models for Codesign (MEMOCODE'11)*, pages 31–42. IEEE, 2011. (Cited on page 154.)

[112] Georg Hofferek and Ashutosh Gupta. Suraq - A controller synthesis tool using uninterpreted functions. In *Haifa Verification Conference (HVC'14)*, volume 8855 of *Lecture Notes in Computer Science*, pages 68–74. Springer, 2014. (Cited on page 154.)

[113] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *Formal Methods in Computer-Aided Design (FMCAD'13)*, pages 77–84. IEEE, 2013. (Cited on pages 154 and 165.)

[114] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *International Conference on Software Engineering (ICSE'94)*, pages 191–200. IEEE/ACM, 1994. (Cited on page 136.)

[115] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004. (Cited on pages 16, 22 and 23.)

[116] Swen Jacobs and Roderick Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10(1), 2014. (Cited on page 154.)

[117] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition. *STTT*. To appear. Preprint available at `http://arxiv.org/abs/1506.08726`. (Cited on pages 3, 7, 24, 82, 83, 84, 86 and 155.)

[118] Swen Jacobs, Viktor Kuncak, and Philippe Suter. Reductions for synthesis procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, volume 7737 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2013. (Cited on page 158.)

[119] Dietmar Jannach and Ulrich Engler. Toward model-based debugging of spreadsheet programs. In *Joint Conference on Knowledge-Based Software Engineering (JCKBSE'10)*, pages 252–264, 2010. (Cited on page 156.)

[120] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In *Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012. (Cited on pages 20, 83, 88 and 154.)

[121] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Automated Software Engineering (ASE'09)*, pages 662–664. IEEE, 2009. (Cited on page 156.)

[122] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012. (Cited on page 19.)

[123] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. (Cited on page 106.)

[124] Jie-Hong Roland Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large boolean relations. In *International Conference on Computer-Aided Design (ICCAD'09)*, pages 779–784. IEEE, 2009. (Cited on pages xvii, 75, 76, 77 and 79.)

[125] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design (FMCAD'06)*, pages 117–124. IEEE, 2006. (Cited on pages 85 and 155.)

[126] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 258–262. Springer, 2007. (Cited on page 155.)

[127] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005. (Cited on page 159.)

[128] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 78(2):441–460, 2012. (Cited on pages 4 and 159.)

[129] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE'02)*, pages 467–477. ACM, 2002. (Cited on page 156.)

[130] Manu Jose and Rupak Majumdar. Bug-Assist: Assisting fault localization in ANSI-C programs. In *Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011. (Cited on pages 8, 141, 142, 157 and 160.)

[131] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Programming Language Design and Implementation (PLDI'11)*, pages 437–446. ACM, 2011. (Cited on pages 106, 141, 157 and 160.)

[132] Ulrich Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Conference on Artificial Intelligence (AAAI'04)*, pages 167–172. AAAI Press / The MIT Press, 2004. (Cited on page 114.)

[133] Hadi Katebi, Karem A. Sakallah, and João P. Marques Silva. Empirical study of the anatomy of modern SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT'11)*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2011. (Cited on page 19.)

[134] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. PARTY parameterized synthesis of token rings. In *Computer Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 928–933. Springer, 2013. (Cited on page 154.)

[135] Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis (ISSTA'15)*, pages 165–176. ACM, 2015. (Cited on page 160.)

[136] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. (Cited on pages 1, 7 and 34.)

[137] Patrick Klampfl. Efficient circuit synthesis using interpolation. Bachelor's thesis, Graz University of Technology, 2014. (Cited on page 41.)

[138] Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In Biere et al. [25], pages 735–760. (Cited on pages 14 and 20.)

[139] Robert Könighofer. Debugging formal specifications with simplified counterstrategies. Master's thesis, Graz University of Technology, 2009. (Cited on page 10.)

[140] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD'11)*, pages 91–100. IEEE, 2011. (Cited on pages 4, 9, 13, 103 and 153.)

[141] Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *Haifa Verification Conference (HVC'12)*, volume 7857 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2012. (Cited on pages 9, 13, 103 and 153.)

[142] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design (FMCAD'09)*, pages 152–159. IEEE, 2009. (Cited on page 10.)

[143] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Haifa Verification Conference (HVC'10)*, volume 6504 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2010. (Cited on pages 10 and 156.)

[144] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013. (Cited on page 10.)

[145] Robert Könighofer, Ronald Toegl, and Roderick Bloem. Automatic error localization for software using deductive verification. In *Haifa Verification Conference (HVC'14)*, volume 8855 of *Lecture Notes in Computer Science*, pages 92–98. Springer, 2014. (Cited on pages 9, 13, 103 and 153.)

[146] Konstantin Korovin. iProver - An instantiation-based theorem prover for first-order logic (system description). In *International Joint Conference on Automated Reasoning (IJCAR'08)*, volume

5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008. (Cited on pages 22 and 83.)

[147] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 430–433. Springer, 2010. (Cited on page 158.)

[148] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Programming Language Design and Implementation (PLDI'10)*, pages 316–329. ACM, 2010. (Cited on page 158.)

[149] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science (FOCS'05)*, pages 531–542. IEEE, 2005. (Cited on page 155.)

[150] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. Evaluation of measures for statistical fault localisation and an optimising scheme. In *Fundamental Approaches to Software Engineering (FASE'15)*, volume 9033 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2015. (Cited on page 156.)

[151] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jonathan D. Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004. (Cited on page 107.)

[152] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE, 2004. (Cited on page 35.)

[153] Harry R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980. (Cited on pages 6, 17 and 22.)

[154] Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF solving. In *Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2008. (Cited on page 21.)

[155] Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7(2-3):71–76, 2010. (Cited on pages 20, 21, 69 and 83.)

[156] Florian Lonsing and Uwe Egly. Incremental QBF solving. In *Principles and Practice of Constraint Programming (CP'14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2014. (Cited on pages 21, 46 and 74.)

[157] Florian Lonsing and Uwe Egly. Incremental QBF solving by DepQBF. In *International Congress on Mathematical Software (ICMS'14)*, volume 8592 of *Lecture Notes in Computer Science*, pages 307–314. Springer, 2014. (Cited on pages 74 and 83.)

[158] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Model-based debugging of Java programs. In *Workshop on Automated Debugging (AADEBUG'00)*, 2000. (Cited on page 156.)

[159] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003. (Cited on pages 1 and 94.)

[160] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Design Automation Conference (DAC'90)*, pages 52–57. IEEE, 1990. (Cited on page 18.)

[161] In-Ho Moon, James H. Kukula, Thomas R. Shiple, and Fabio Somenzi. Least fixpoint approximations for reachability analysis. In *International Conference on Computer-Aided Design (ICCAD'99)*, pages 41–44. IEEE, 1999. (Cited on pages 57 and 165.)

[162] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. (Cited on pages 8 and 122.)

[163] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. Solving games using incremental induction. In *Integrated Formal Methods (IFM'13)*, volume 7940 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2013. (Cited on pages 8, 67, 86, 88, 91, 93 and 153.)

[164] Glenford J. Myers. *The Art of Software Testing*. Wiley, 3rd edition, 2011. (Cited on page 1.)

[165] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer-Aided Design (FMCAD'10)*, pages 221–229. IEEE, 2010. (Cited on page 14.)

[166] Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk, and Adam Walker. Solving games without controllable predecessor. In *Computer Aided Verification (CAV'14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 533–540. Springer, 2014. (Cited on page 153.)

[167] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002. (Cited on page 132.)

[168] Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF (tool presentation). In *Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 430–435. Springer, 2012. (Cited on pages 6 and 68.)

[169] 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. (Cited on page 22.)

[170] Philipp Pani. Implementation of a symbolic execution engine for a software debugging tool. Bachelor's thesis, Graz University of Technology, 2011. (Cited on page 132.)

[171] Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986. (Cited on page 27.)

[172] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990. (Cited on page 139.)

[173] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *Foundations of Computer Science (FOCS'79)*, pages 348–363. IEEE, 1979. (Cited on page 65.)

[174] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. (Cited on pages 14, 45, 46, 53, 66 and 74.)

[175] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, 1977. (Cited on pages 3, 29 and 154.)

[176] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL'89)*, pages 179–190. ACM, 1989. (Cited on page 3.)

[177] Steven David Prestwich. CNF encodings. In Biere et al. [25], pages 75–97. (Cited on page 19.)

[178] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. (Cited on page 1.)

[179] Jaan Raik, Urmas Repinski, Anton Chepurov, Hanno Hantson, Raimund Ubar, and Maksim Jenihhin. Automated design error debug using high-level decision diagrams and mutation operators. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(4-5):505–513, 2013. (Cited on pages 106 and 159.)

[180] Silvio Ranise and Cesare Tinelli. Satisfiability modulo theories. *Trends and Controversies-IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006. (Cited on page 22.)

[181] Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Algorithms for omega-regular games with imperfect information. *Logical Methods in Computer Science*, 3(3), 2007. (Cited on page 155.)

[182] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987. (Cited on pages 7, 38, 39, 40, 45, 113, 114, 115, 132, 156 and 157.)

[183] Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *European Test Symposium (ETS'12)*, pages 1–6. IEEE, 2012. (Cited on page 159.)

[184] Thomas W. Reps, Thomas Ball, Manuvir Das, and James R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *European Software Engineering Conference held jointly with Foundations of Software Engineering (ESEC/FSE'97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 432–449. Springer, 1997. (Cited on page 156.)

[185] Heinz Riener, Rüdiger Ehlers, and Görschwin Fey. Path-based program repair. In *Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA'15)*, volume 178 of *EPTCS*, pages 22–32, 2015. (Cited on page 158.)

[186] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, February 1992. (Cited on pages 3 and 30.)

[187] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design (ICCAD'93)*, pages 42–47. IEEE, 1993. (Cited on pages 18 and 93.)

[188] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *Operating Systems Design and Implementation (OSDI'14)*, pages 661–676. USENIX Association, 2014. (Cited on page 85.)

[189] Martina Seidl and Robert Könighofer. Partial witnesses from preprocessed quantified boolean formulas. In *Design, Automation & Test in Europe (DATE'14)*, pages 1–6. IEEE, 2014. (Cited on pages 8, 9, 10, 21, 46, 74, 83 and 165.)

[190] Martina Seidl, Florian Lonsing, and Armin Biere. qbf2epr: A tool for generating EPR formulas from QBF. In *Practical Aspects of Automated Reasoning (PAAR'12)*, volume 21 of *EPiC Series*, pages 139–148. EasyChair, 2012. (Cited on page 65.)

[191] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference held jointly with Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005. (Cited on pages 2, 7 and 36.)

[192] Detlef Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, 2002. (Cited on page 18.)

[193] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design (ICCAD'96)*, pages 220–227. IEEE/ACM, 1996. (Cited on page 19.)

[194] Michael Sipser. *Introduction to the theory of computation*. Course Technology, Cengage Learning, Boston, Massachusetts, 2nd edition, 2008. (Cited on page 33.)

[195] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013. (Cited on page 24.)

[196] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. (Cited on pages 7, 8, 32, 119, 144, 150, 151 and 157.)

[197] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. (Cited on pages 4, 7, 8, 32, 119, 144, 150, 151 and 157.)

[198] Fabio Somenzi. CUDD: BDD package, University of Colorado, Boulder. `http://vlsi.colorado.edu/~fabio/CUDD/`. (Cited on pages 18 and 93.)

[199] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000. (Cited on page 155.)

[200] Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bit-vector arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 2012. (Cited on page 158.)

[201] Stefan Staber and Roderick Bloem. Fault localization and correction with QBF. In *Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 355–368. Springer, 2007. (Cited on pages 41 and 153.)

[202] Markus Stumptner and Franz Wotawa. Debugging functional programs. In *International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1074–1079. Morgan Kaufmann, 1999. (Cited on page 156.)

[203] Geoff Sutcliffe and Christian B. Suttner. The TPTP problem library - CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. (Cited on page 22.)

[204] Geoff Sutcliffe and Christian B. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006. (Cited on page 22.)

[205] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995. (Cited on pages 26 and 27.)

[206] Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .NET. In *Tests and Proofs (TAP'08)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. (Cited on page 36.)

[207] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995. (Cited on page 157.)

[208] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983. (Cited on pages 14, 60, 66, 72 and 77.)

[209] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. (Cited on page 33.)

[210] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. *STTT*, 15(5-6):413–431, 2013. (Cited on page 160.)

[211] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE'81)*, pages 439–449. IEEE, 1981. (Cited on page 157.)

[212] Andreas Zeller. *Why programs fail - a guide to systematic debugging*. Elsevier, 2006. (Cited on page 1.)

[213] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. (Cited on pages 114, 157 and 160.)

Index